WEBOTS 2.0
User Guide

© 1998, 1999 Cyberbotics
www.cyberbotics.com

July 13, 1999

© 1998, 1999 Cyberbotics S.a r.l.
All Rights Reserved

CYBERBOTICS S.A R.L. (“CYBERBOTICS”) MAKES NO WARRANTY OR CONDITION, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THIS MANUAL AND THE ASSOCIATED SOFTWARE. THIS MANUAL IS PROVIDED ON AN “AS-IS” BASIS. NEITHER CYBERBOTICS NOR ANY APPLICABLE LICENSOR WILL BE LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES.

This software was initially developped at the Laboratoire de Micro-Informatique (LAMI) of the Swiss Federal Institute of Technology, Lausanne, Switzerland (EPFL). The EPFL makes no warranties of any kind on this software. In no event shall the EPFL be liable for incidental or consequential damages of any kind in connection with use and exploitation of this software.

Trademark information

IRIX, O2 and OpenGL are registered trademark of Silicon Graphics Inc.

Khepera is a registered trademark of K-Team S.A.

Linux is a registered trademark of Linus Torwalds.

Macintosh is a registered trademark of Apple Computer, Inc.

Pentium is a registered trademark of Intel Corp.

PowerPC is a registered trademark of IBM Corp.

Red Hat is a registered trademark of Red Hat Software, Inc.

Solaris and Solaris OpenGL are registered trademarks of Sun Microsystems, Inc.

All SPARC trademarks are used under the license and are trademarks or registered trademarks of SPARC International, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

SuSE is is a registered trademark of SuSE GmbH.

UNIX is a registered trademark licensed exclusively by X/Open Company, Ltd.

Foreword

Webots was originally developed as a research tool for investigating various control algorithms in mobile robotics. This simulation software has been carefully designed to facilitate the transfer to real robots. It is currently specific to the Khepera robot and to the Alice robot. However, forthcoming versions of Webots will include more robots.

This user guide will get you started using Webots. However, the reader is expected to have a minimal knowledge in mobile robotics as well as in C programming. It is not necessary to have a real robot to use Webots.

Cyberbotics is grateful to all the people who contributed to the development of Webots, Webots sample applications, Webots User Guide, Webots Reference Manual, and the Webots web site, including Yuri Lopez de Meneses, Auke-Jan Ijspeert, Gerald Foliot, Allen Johnson, Michael Kertesz, Aude Billiard, and many others. Moreover, many thanks are due to Prof. J.-D. Nicoud (LAMI-EPFL) and Dr. F. Mondada for their valuable support.

Contents

1 Installing Webots **13**

1.1 Hardware requirements 13

1.2 Registration procedure 13

1.2.1 Principle 13

1.2.2 Registering 14

1.3 Installation procedure 15

1.3.1 PC i386 / Linux 15

1.3.2 Macintosh PowerPC / MkLinux or Linux PPC 16

1.3.3 Sun Sparc / Solaris 16

2 Webots Basics **19**

2.1 Running Webots 19

2.2 Running a simulation 19

2.3 Exporting as animated GIF 21

2.4 Controlling the point of view 21

2.4.1 Navigating in the scene 22

2.4.2 Switching between the World View and the Robot View 22

2.5 Editing the environment 22

2.5.1 Creating, opening and saving worlds 22

2.5.2 Adding objects 23

2.5.3 Moving objects 25

2.5.4 Cut, copy and paste operations 26

2.5.5 Changing the Background 27

2.5.6 Changing the Ground . . . 27
2.5.7 Going further . . . 28
2.6 Working with the Khepera robots . . . 28
2.6.1 Khepera overview . . . 28
2.6.2 Creating a simulated Khepera . . . 29
2.6.3 Moving the Khepera . . . 30
2.6.4 Sensor representation . . . 31
2.6.5 Motors representation . . . 32
2.6.6 Jumper configuration . . . 32
2.6.7 Extension turrets: K-Bus . . . 32
2.6.8 Khepera controllers . . . 33
2.6.9 Moving to the real Khepera via the serial connection . . . 33
2.7 Working with Supervisors . . . 36
2.8 Working with the Alice robot . . . 37
2.8.1 Alice overview . . . 37
2.8.2 Programming the Alice robot . . . 37
2.9 Miscellaneous . . . 38
2.9.1 Hiding and showing the buttons . . . 38
2.9.2 Setting up preferences . . . 38
2.10 Directory structure . . . 39

3 Sample Webots Applications 41

3.1 simple.wbt . . . 41
3.2 five.wbt . . . 42
3.3 phototaxy.wbt . . . 43
3.4 jumper.wbt . . . 44
3.5 finder.wbt . . . 45
3.6 can.wbt . . . 46
3.7 attacker.wbt . . . 47
3.8 buffer.wbt . . . 48
3.9 town.wbt . . . 49

3.10 house.wbt 50
3.11 chase.wbt 51
3.12 stick_pulling.wbt 52
3.13 alice.wbt 53

4 Programming with the Khepera API **55**
4.1 My first Khepera program 55
4.1.1 Source code 55
4.1.2 Controller directory structure 56
4.1.3 Compiling the controller 56
4.1.4 Modifying the controller 56
4.2 Webots execution scheme 57
4.2.1 Khepera controllers 57
4.2.2 Other controllers 58
4.3 Getting sensor information 59
4.4 Controlling Actuators 60
4.5 Working with extension turrets 60
4.5.1 K213 turret 61
4.5.2 K6300 turret 61
4.5.3 Gripper turret 61
4.5.4 Panoramic turret 62

5 Programming with the Supervisor API **71**
5.1 EAI: External Authoring Interface 71
5.1.1 Getting a pointer to a node 71
5.1.2 Reading information from the world 72
5.1.3 Editing the world 72
5.1.4 Sending messages to the robots 72

6 Using GUI: the Graphical User Interface API **75**

6.1 Basics . . . 75

6.1.1 Include file and library . . . 75

6.1.2 GUI objects . . . 75

6.1.3 Constants . . . 76

6.1.4 Functions . . . 76

6.2 Getting started . . . 78

6.3 Editing, adding and deleting gobs . . . 78

6.3.1 Editing gobs . . . 78

6.3.2 Adding gobs . . . 79

6.3.3 Deleting gobs . . . 79

6.4 Working with widgets . . . 80

6.4.1 Events . . . 80

6.4.2 Callback function . . . 81

6.5 Going further . . . 82

7 Advanced Webots programming **83**

7.1 Hacking the world files . . . 83

7.1.1 Overview . . . 83

7.1.2 Robots and Supervisors . . . 83

7.1.3 Textures . . . 84

7.2 Using external C/C++ libraries . . . 84

7.3 Interfacing with third party software . . . 84

7.3.1 Using a pipe based interface . . . 84

7.3.2 Using other Inter Process Communication systems . . . 85

8 Troubleshooting **87**

8.1 Common problems and solutions . . . 87

8.2 How to I send a bug report ? . . . 88

List of Figures

1.1 Webots registration page 14
1.2 Webots popup window 15

2.1 Default world 20
2.2 Speedometer 21
2.3 Creating a new world 23
2.4 Can properties window 24
2.5 Lamp properties window 25
2.6 Wall properties window 26
2.7 Vertice ordering in Wall nodes 27
2.8 Background color 27
2.9 Ground properties 28
2.10 The Khepera robot 29
2.11 Creating a new robot 29
2.12 Khepera properties window 30
2.13 Khepera bus plugin 33
2.14 Successful serial connection 35
2.15 Downloading a cross-compiled controller 36
2.16 The Alice robot 37
2.17 Webots preferences 38
2.18 Webots directory structure 39

3.1 five.wbt 42
3.2 phototaxy.wbt 43
3.3 jumper.wbt 44

3.4 finder.wbt . . . 45
3.5 can.wbt . . . 46
3.6 attacker.wbt . . . 47
3.7 buffer.wbt . . . 48
3.8 town.wbt . . . 49
3.9 house.wbt . . . 50
3.10 chase.wbt . . . 51
3.11 stick_pulling.wbt . . . 52
3.12 alice.wbt . . . 53

4.1 Structure of a controller program . . . 58
4.2 The real K213 turret . . . 63
4.3 The simulated K213 turret . . . 64
4.4 The robot window for the K213 turret . . . 64
4.5 The real K5300/K6300 turret . . . 65
4.6 The simulated K6300 turret . . . 66
4.7 The robot window for the K6300 turret . . . 66
4.8 The real gripper turret . . . 67
4.9 The simulated gripper turret . . . 68
4.10 The robot window for the gripper turret . . . 68
4.11 The real panoramic turret (prototype) . . . 69
4.12 The simulated panoramic turret . . . 70
4.13 The robot window for the panoramic turret . . . 70

5.1 Command byte for the Alice robot . . . 74
5.2 Parameter byte for the Alice robot . . . 74

6.1 Widgets available in the GUI . . . 80

List of Tables

1.1 Supported system configurations 13

2.1 Degrees of freedom for navigation 22
2.2 Khepera sensor values 31

5.1 Alice Messages 73

Chapter 1

Installing Webots

1.1 Hardware requirements

Webots is supported on the UNIX / X11 systems described in table 1.1:

Supported Hardware	Recommanded Hardware	Operating System
PC i386	Pentium II	Linux with glibc RedHat 6.0, SuSE 6.1 or Debian 2.1
Macintosh PowerPC	PowerPC G3	Linux PPC R5 or MkLinux DR3
Sun Sparc Workstation	Sparc 20	Solaris 2.6
SGI Workstation	any	Irix 6.2

Table 1.1: Supported system configurations

OpenGL hardware acceleration is supported only on Sun with 3D graphics board and SGI workstations.

1.2 Registration procedure

1.2.1 Principle

In order to be able to run Webots, you need a license file called `webots.key` which will enable you to use Webots according to the license you bought. Several licenses are available which allow Webots to run on a single computer or on a specified number of computers. Hence, a single computer license is specific to one computer, and a multiple computer license is specific to a number of computers as specified in the licence. For this reason, we recommend that you be very careful when providing us data relative to your computer(s) because this data cannot

be changed in the future: **IT WILL NOT BE POSSIBLE TO CHANGE THE HOSTNAME AND HOSTID OF THE REGISTERED COMPUTER**.

Please read your license agreement carefully before registering. This license is provided within the software package. By using the software and documentation, you agree to abide by all the provisions of this license.

1.2.2 Registering

In order to register your computer(s) to run Webots, you will have to fill out a form on the Web and you will receive via e-mail the `webots.key` file corresponding to your license. The form is available at the following web address:

`http://www.cyberbotics.com/registration/webots.html` (see figure 1.1)

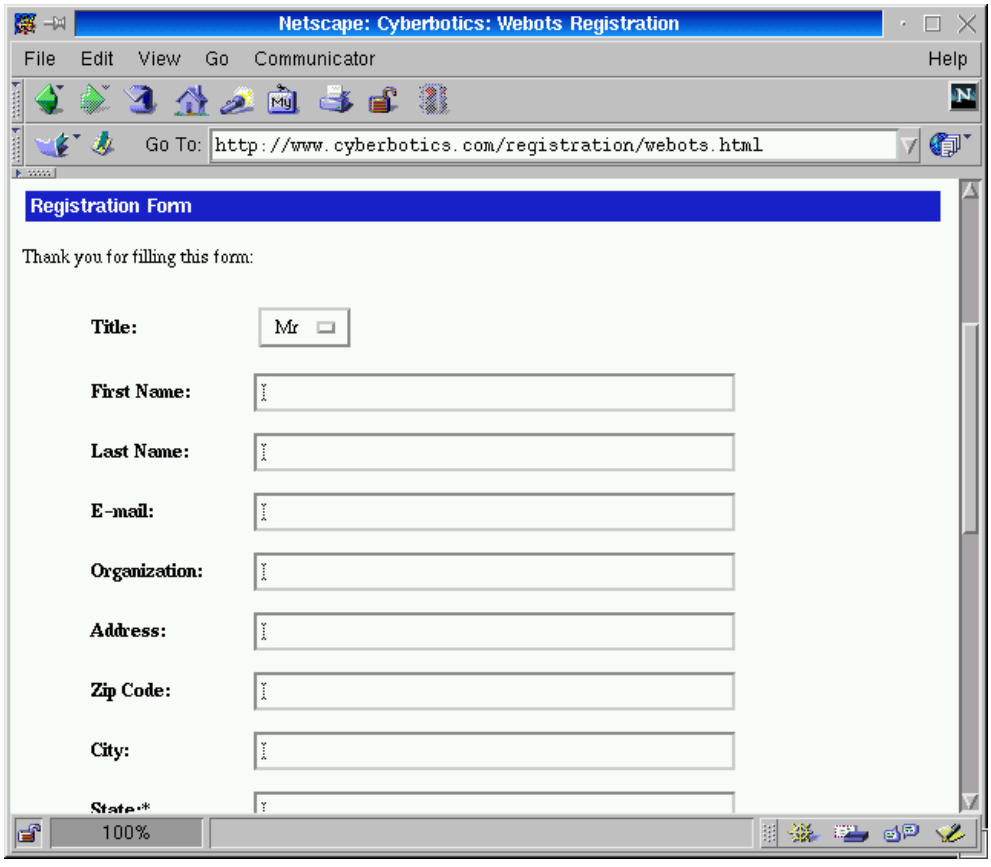

Figure 1.1: Webots registration page

Please take care to properly fill each field of this form. The `Serial Number` is the serial number of your own Webots package which is printed on the back side of the CD-ROM package under the heading `S/N:`.

The `hostname` and `hostid` fields are specific to your computer. You can get this information from your computer by launching Webots without a `webots.key` file installed. The first window appearing will display your `hostname` and `hostid` (see figure 1.2). A typical `hostname` would be `lamipc2.epfl.ch` and a typical `hostid` would be `0x5fcbec77`.

Figure 1.2: Webots popup window

If your license allows you to run Webots on several computers (like Lab Pack 5, Lab Pack 10 or Lab Pack 20 licenses), then you will have to enter the complete list of `hostname`s separated by spaces as well as the corresponding list of `hostid`s also separated by spaces. Take care that the same order is respected for both lists, that is, the first `hostname` must correspond to the first `hostid`, the second `hostname` must correspond to the second `hostid`, etc.

After completing this form, click on the `Submit` button. Then, you will receive, within a few hours, via e-mail to the address you mentioned in the form, your personal `webots.key` file, which is needed to install a registered version of Webots on your system as we will see below.

1.3 Installation procedure

In order to install Webots on your system, you will have to follow the instructions corresponding to your computer / operating system listed below:

1.3.1 PC i386 / Linux

1. Log on as `root`.
2. Insert the CD-ROM and mount it (this might be automatic).
   `mount /mnt/cdrom`

3. Install the RedHat packages:
   `rpm -Uvh /mnt/cdrom/convert-1.0-1.i386.rpm` (optional: useful to export Webots simulations towards animated gifs)
   `rpm -Uvh /mnt/cdrom/Mesa-3.0-1.i386.rpm`
   `rpm -Uvh /mnt/cdrom/webots-2.0-1.i386.rpm`

4. You may need to set the environment variable `WEBOTS_HOME` to `/usr/local/webots`. This can be achieved from one of the following commands:
   `export WEBOTS_HOME=/usr/local/webots`
   `setenv WEBOTS_HOME /usr/local/webots`
   depending on your shell. Note that the rpm installation script will automatically set this variable for you in the `/etc/profile` file, so that any user of your system could launch Webots.

5. Copy your personal `webots.key` file into the `/usr/local/webots/` directory.

1.3.2 Macintosh PowerPC / MkLinux or Linux PPC

The installation procedure is the same as for the PC i386 / Linux system, except that you must install the RedHat packages `convert-1.0-1.ppc.rpm`, `Mesa-2.6-1a.ppc.rpm` and `webots-2.0-1.ppc.rpm` corresponding to the Linux PowerPC architecture.

1.3.3 Sun Sparc / Solaris

1. Log on as `root`.

2. Check whether the OpenGL library is installed:
   `what /usr/lib/libGL.so`
   If it is installed you should get a message indicating the release version. The most recent version is 1.2 (as of December 18, 1998). If it is not installed or you would like to get the latest release, you might want to go to
   `http://www.sun.com/solaris/opengl/`
   and follow the instructions to get it installed.

3. Insert the CD-ROM and mount it. If you have Sun OpenGL installed, copy the file `webots2.0Solaris.tar.gz` from the CD-ROM to your `/usr/local/` directory. This directory is taken as an example and can be changed according to the organization of your file system. If you don't want to install Sun OpenGL, you can use the package `webots2.0SolarisMesa.tar.gz` instead of `webots2.0Solaris.tar.gz`.

4. Execute: `cd /usr/local`

5. Uncompress the package: `gunzip webots2.0Solaris.tar.gz`

6. Untar the package: `tar xvf webots2.0Solaris.tar`
7. Execute one of the following commands:
   `export WEBOTS_HOME=/usr/local/webots`
   `setenv WEBOTS_HOME /usr/local/webots`
   depending on your shell.
8. Create a shortcut so that the `webots` executable is in the PATH of any user:
   `ln -s /usr/local/webots/webots /usr/local/bin/webots`
9. Set the environment variable WEBOTS_HOME of the users who want to use Webots to `/usr/local/webots` as you did in (7).
10. Copy your personnal `webots.key` file into the `/usr/local/webots/` directory.

Chapter 2

Webots Basics

2.1 Running Webots

Type `webots` to launch the simulator. You should see the world window appear on the screen (see figure 2.1).

From there, a number of possibilities are available through menus and buttons. The control wheels and their associated buttons on the left, right and bottom side of the window allow you to navigate in the scene. You can also click in the scene to select an object (`Alice`, `Ball`, `Can`, `Ground`, `Khepera`, `Lamp` or `Wall`) or double-click on an object to open its properties window.

The "About" window is available from the `About...` item in the `File` menu. This window displays the license information.

To quit Webots, you can either close the world window, or use the `Quit` item in the `File` menu.

2.2 Running a simulation

In order to run a simulation, a number of buttons are available, which correspond to menu items in the `Simulation` menu:

- `Stop`: interrupt the `Run` or the `Fast` mode.
- `Step`: execute one simulation step.
- `Step`: turn on/off the recording mode (to export the simulation to an animated GIF image).

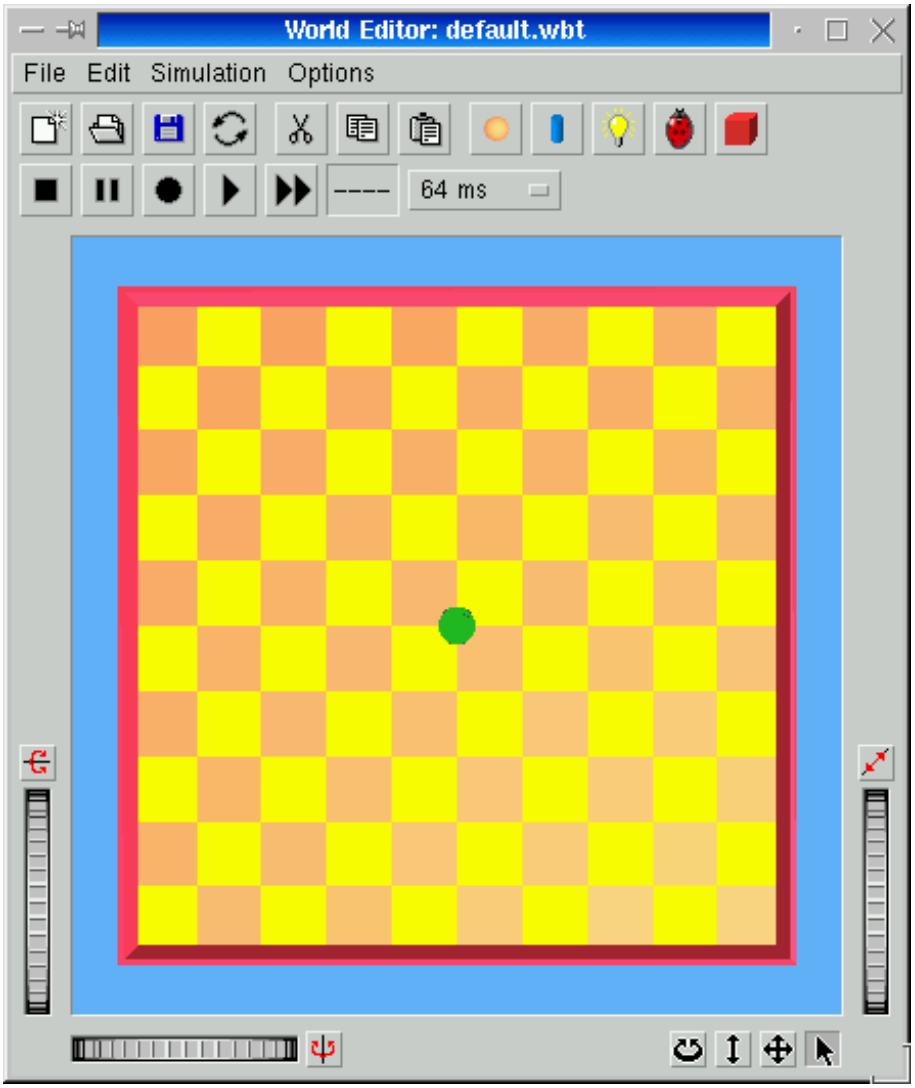

Figure 2.1: Default world

- `Run`: execute simulation steps until the `Stop` mode is entered.
- `Fast`: same as `Run`, except that no display is performed.

The `Fast` mode corresponds to a very fast simulation mode suited for heavy computation (genetic algorithms, vision, learning, etc.). However, since no display of the world is performed during `Fast` simulation, the scene in the world window becomes black until the `Fast` mode is stopped.

A speedometer (see figure 2.2) allows you to observe the speed of simulation on your computer. It indicates how fast the simulation runs comparing to real time. In other words, it represents

the speed of the virtual time. If the value of the speedometer is 2, it means that your computer simulation is running twice as fast as the corresponding real robots would do. This information is relevant in `Run` mode as well as in `Fast` mode.

Figure 2.2: Speedometer

The time step can be choosen from the popup menu situated next to the speedometer. It indicates how frequently the display is refreshed. It is expressed in virtual time milliseconds. The value of this time step also defines the duration of the time step executed during the `Step` mode.

In `Run` mode, with a time step of 64 ms and a fairly simple world displayed with the default window size, the speedometer will typically indicate approximately 0.5 on a Pentium II / 266 without hardware acceleration and 4 on an Ultra Sparc 10 Creator 3D.

2.3 Exporting as animated GIF

The `Record` mode will allow you to export a simulation as an animated GIF file that can further be used to set up nice web pages or multi-media presentations. In order to proceed, you will need to have the convert utility installed (see intallation procedure in this manual). Webots will save each image in PNG format in the temporary folder. Then, it will invoque the convert program to produce the animated GIF file from the PNG images. However, note that the animated GIF files produced here are not optimized and hence can be huge! So, reduce the size of the view as much as possible, and don't make too long movies... Actually, each image is stored in the GIF file whereas it would be possible to optimze it by storing only the changes from one image frame to the other. In order to proceed such an optimization we recommand you to use the Gimp software (GNU Image Manupulation Program) which is a very powerful free software tool. This software is included on the Webots CD-ROM and can be used under the terms of the GNU General Public License.

2.4 Controlling the point of view

The view of the scene is produced by a camera which is set according to a given position and orientation. You can change this position and orientation to navigate in the scene. Moreover you can set the camera to follow the movements of a given robot or to remain fixed.

2.4.1 Navigating in the scene

To navigate in the 3D scene, you can use the three control wheels situated on the edges of the world window. Each wheel is associated with a button allowing to switch between a rotation and a translation. Hence, you can control six degrees of freedom to move the camera in the scene. These degrees of freedom are also available from the keyboard when the world window is active as explained in table 2.1.

axis	Rotation	modifier	keys	Translation	modifier	keys
X	tilt	shift	$\leftarrow \rightarrow$	horizonal scroll	ctrl	$\uparrow \downarrow$
Y	pan	shift	$\leftarrow \rightarrow$	vertical scroll	ctrl	$\uparrow \downarrow$
Z	roll		$\leftarrow \rightarrow$	zoom		$\uparrow \downarrow$

Table 2.1: Degrees of freedom for navigation

Notice that if an object is selected in the scene, the camera will rotate, according to the selected axis, around this object.

2.4.2 Switching between the World View and the Robot View

You can choose between two different points of view:

- `World View`: this view corresponds to a fixed camera standing in the world.
- `Robot View`: this view corresponds to a mobile camera following a robot.

You can switch between both modes using the last item of the `Simulation` menu. But before you go into this menu, you have to select a robot if you want a `RobotView` from this robot, or unselect any robot if you want a `World View` (the most convenient way to unselect any robot is to click on an object or in the background of the scene). Then, the last item of the `Simulation` menu will be set appropriately, so that you can select either the `Robot View` or the `World View`.

2.5 Editing the environment

2.5.1 Creating, opening and saving worlds

When the simulator is launched, a default world is loaded in the world window. In order to create your own world, you can go to the `File` menu and select the `New...` item. A popup window will appear as depicted on figure 2.3 letting you choose the size of the ground grid and the number of plates it is made of.

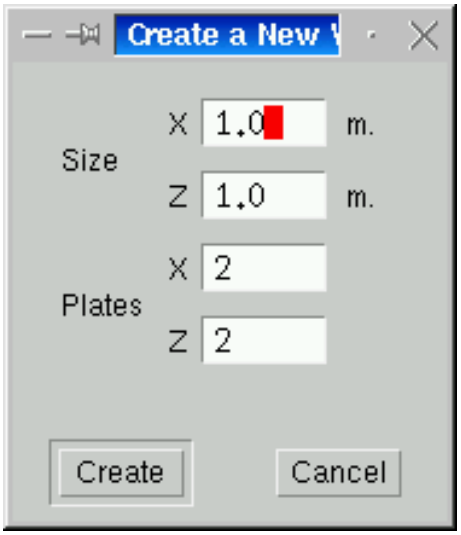

Figure 2.3: Creating a new world

If one of these fields is set to zero, no ground is created.

The `File` menu will also allow you to perform the standard file operations: `Open..., Save` and `Save As...`, respectively, to load, save and save with a new name the current world.

The following buttons can be used as shortcuts to menu items:

- `New...`
- `Open...`
- `Save`

The default world directory contains a lot of examples of worlds files (files ending up with the `.wbt` suffix) which are described in chapter 3. However, you won't be able to save your own worlds in this directory unless you are logged on as `root`. We do not recommend that you do so, however. Instead, you should create a `worlds` directory in your home directory and save your own world files there.

2.5.2 Adding objects

Several kinds of objects can be added from the world window. They are available through the `Edit` menu or the corresponding buttons:

- `New Ball...` a ball with which the robots can play.
- `New Can...` a cylinder that can be grasped by the Khepera robots equipped with a gripper turret.
- `New Lamp...` a light bulb emitting light visible by the infra-red sensors of the Khepera robot.
- `New Robot...` a robot.
- `New Wall...` a rectangular parallelepiped wall, which is typically an obstacle for robots.

For the creation of each of these objects, a popup window will appear, displaying the properties of the object. You will then be able to change the position, orientation, color, etc. of the objects. Usually the coordinates indicate the center of the object, just like in VRML97. However, the `y` coordinate of the `Can` is an exception. It should always be set to 0, indicating that the can is set on the ground. Here are some examples of properties windows depicted on figures 2.4, 2.5 and 2.6.

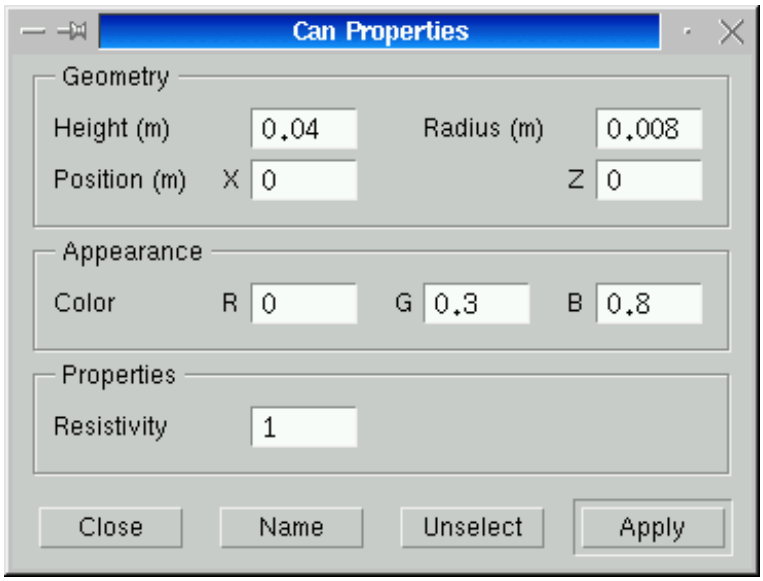

Figure 2.4: Can properties window

The `Resistivity` parameter of the `Can` object corresponds to the electrical resistivity of the object as it can be measured by the sensor of the Gripper turret of the Khepera robot. This value ranges from 0 to 255.

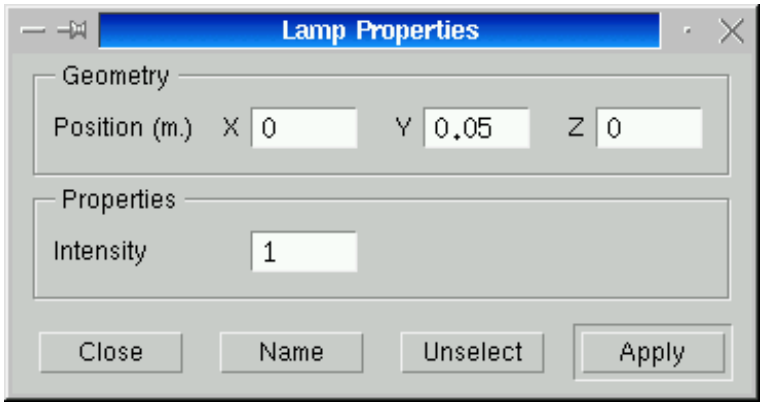

Figure 2.5: Lamp properties window

The `Intensity` parameter of the Lamp object allows one to change the power of the light bulb, so that infra-red sensors are more or less sensitive to this object. A value of 0 means that the lamp is off while a value of 1 means that it is on with the default power. However, unlike with most 3D rendering software, there is no special lighting rendering effects due to the presence of such a `Lamp` in the scene.

The `Wall` is probably one of the most complex object to edit since it contains a variable number of vertices. By default a four-vertices `Wall` is created, but you can add new vertices. You can also remove vertices as long as at least three vertices remain. For each vertice, you can edit its 2D coordinates, so that you define a plane shape that is extruded to produce the `Wall`. Please note that the order of the vertices is important: like in VRML, when progressing in the list of vertices, the right-hand direction (perpendicular to the progression) should always point to the inside of the `Wall` object. The figure 2.7 explains this rule for ordering vertices.

The `height`, `color`, `position` and `orientation` of the `Wall` can be changed by editing the corresponding text fields.

Remember that to display the property window of an object in the scene, you will have to double-click on this object. Robots are special objects. The properties window of robots will be discussed later in this manual.

2.5.3 Moving objects

A number of buttons are available in the bottom right side of the main window. They allow you to select and move objects in the world:

- This turn button allows you to rotate objects: click on this button to enter the turn mode, then click on an object to select it and then click somewhere else and drag the mouse pointer to rotate the selected object.

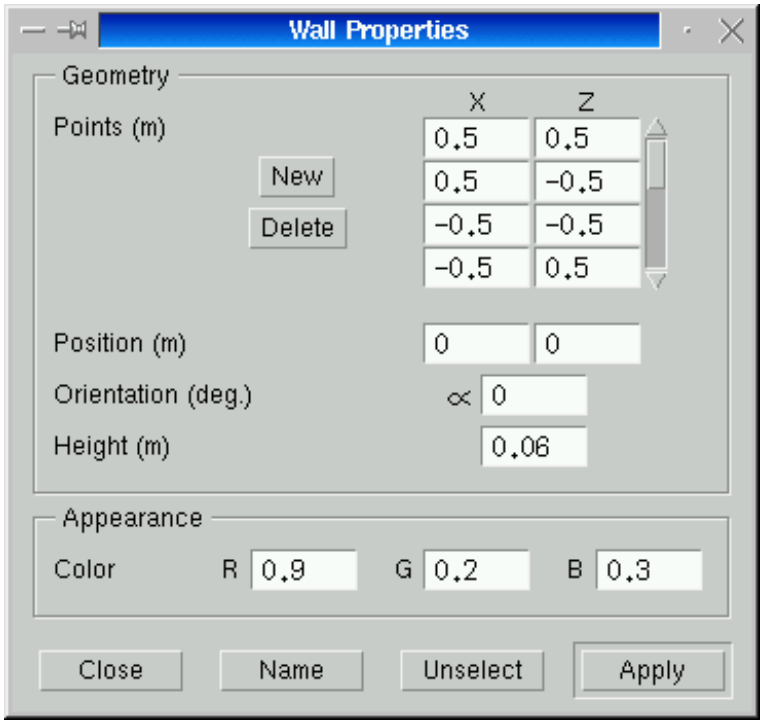

Figure 2.6: Wall properties window

- This lift button allows you to translate objets along the vertical axis. It is especially suited for `Ball` and `Lamp` objects.
- This move button allows you to translate the object on the ground using the familiar drag'n'drop interface. It is probably one of the most useful mode.
- This pointer button allows you to select objects by clicking on them and to open their properties windows by double-clicking on them.

2.5.4 Cut, copy and paste operations

The standard editing operations are avaible either from the `Edit` menu or the following buttons:

- `Cut`
- `Copy`
- `Paste`

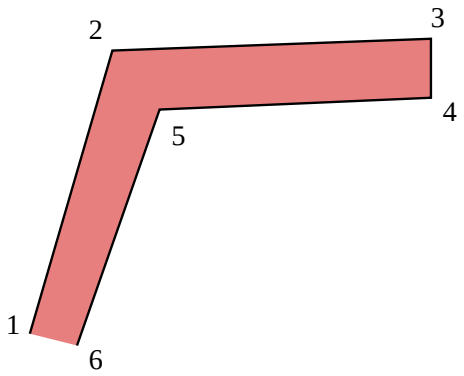

Figure 2.7: Vertice ordering in Wall nodes

In order to perform the `Cut` or `Copy` operation, you first need to select an object by clicking on it in the scene. Pasted objects will appear close to their original clone. You will have to turn, move, or edit their coordinates in their properties windows to move them to the desired location and orientation. Note that robots are special objects which can be cut out, but not copied nor pasted.

2.5.5 Changing the Background

The `Background...` item in the `Edit` menu will allow you to define the color of the background in the world (see figure 2.8). This might be useful, especially when using cameras, to improve distinction between objects and the background.

Figure 2.8: Background color

2.5.6 Changing the Ground

The `Ground...` item in the `Edit` menu will allow you to redefine the ground (size and number of plates) on which robots evolve. It will raise the properties window depicted in figure 2.9.

The first and second colors refer to the two colors used to create the ground grid. You can change them to fit your needs.

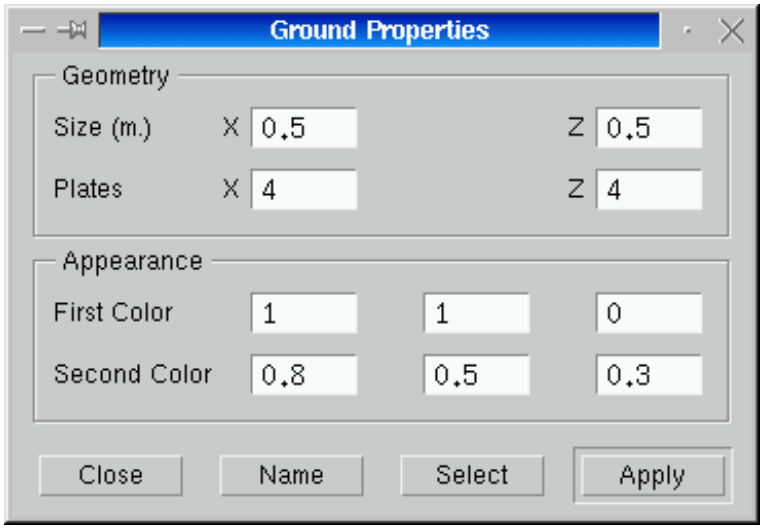

Figure 2.9: Ground properties

2.5.7 Going further

It is also possible in Webots to use a subset of VRML97 nodes. However, these nodes are not available directly from the built-in Webots world editor. Instead, you should use a text editor to edit the world files and add VRML'97 nodes you need. More details about this technique are provided in this manual in the *Avanced Webots Programming* chapter.

2.6 Working with the Khepera robots

2.6.1 Khepera overview

Khepera is a mini mobile robot (5 cm diameter, 70 g.) which was developed at EPFL - LAMI by Andr Guignard, Edoardo Franzi and Francesco Mondada (see figure 2.6.1). It is commercially available from K-Team S.A. (`http://www.k-team.com`). This robot is widespread in many Universities all over the world and is used mainly for research and education. It has two motor wheels and is equiped with 8 infra-red sensors allowing it to detect obstacles all around it in a range of about 5 cm. These sensors can also provide informations about light measurement, allowing the robot to look for light sources. Khepera supports a number of extension turrets including vision turrets, a gripper, a radio turret, a general I/O turret, etc. However, not all of these turrets are implemented in Webots.

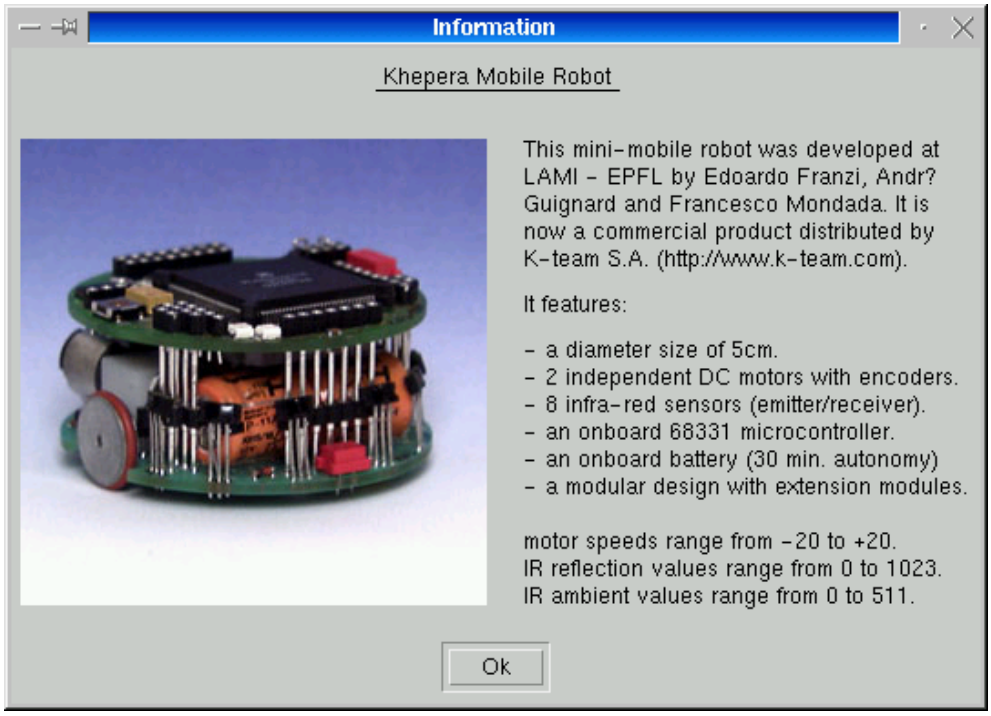
Information
Khepera Mobile Robot
This mini-mobile robot was developed at LAMI - EPFL by Edoardo Franzi, Andr? Guignard and Francesco Mondada. It is now a commercial product distributed by K-team S.A. (http://www.k-team.com).
It features:
- a diameter size of 5cm.
- 2 independent DC motors with encoders.
- 8 infra-red sensors (emitter/receiver).
- an onboard 68331 microcontroller.
- an onboard battery (30 min. autonomy)
- a modular design with extension modules.
motor speeds range from -20 to +20.
IR reflection values range from 0 to 1023.
IR ambient values range from 0 to 511.
Ok

Figure 2.10: The Khepera robot

2.6.2 Creating a simulated Khepera

To create a robot, you can use either the `New Robot...` menu item or the corresponding button as seen previously. A popup menu will let you choose the type of robot you want to create (see figure 2.6.2). Choose Khepera. You will be also prompted to assign a name to the robot. You could choose any name you like. Let's choose *Max*, as an example:

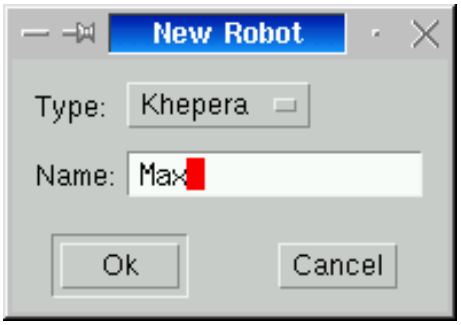

Figure 2.11: Creating a new robot

The properties window of *Max* should then appear (figure 2.6.2. However, it is possible to obtain the properties window of any robot by double-clicking on the desired robot in the world window.

If *Max* isn't a good name for you, you can change this by clicking on the `Name` button. The

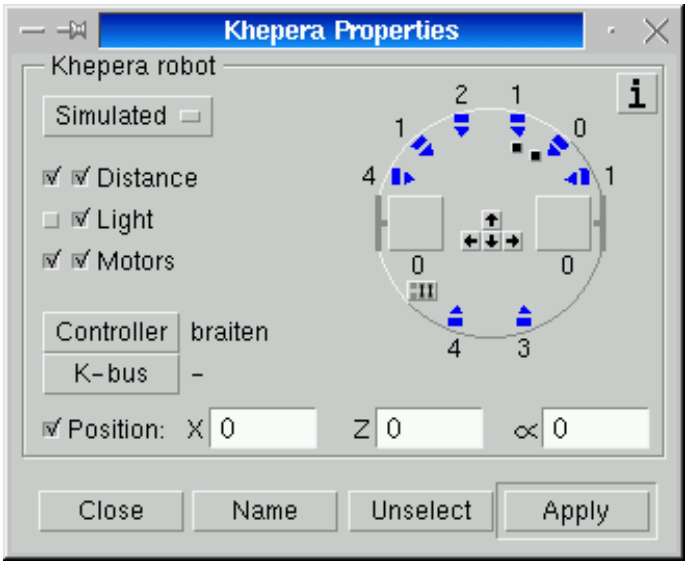

Figure 2.12: Khepera properties window

`Unselect` button can be used to unselect the robot in the world window ; this button will then be changed into a `Select` button allowing the robot to be selected again. This might be very useful when there are several robots in the world and you don't know which one is represented in your robot properties window. The `Apply` button performs the position changes if needed. Finally, the `Close` button allows you to close this window. To get back to it, you will have to double-click on the corresponding robot in the world window.

There is theoretically no limit to the number of robots being simulated. However, the limit is actually the limit of your computer resources (memory, speed, inter-process communications, etc.) and your patience...

2.6.3 Moving the Khepera

Several ways allow you to move the robot. You can either:

- use the buttons for moving / rotating objects.
- click on the little arrow buttons on the robot representation (between both motors).
- press the arrow keys on your keyboard (however, you must be careful that the robot window is the current active window).
- type new coordinates and orientation in the `X`, `Z` and α text fields. `X` and `Z` correspond to the position of the robot on the floor, expressed in meters (m), while α corresponds to the orientation of the robot expressed in radians (rad). You will have to press the enter (or

color	distance measurement	light measurement
blue	no obstacle detected	no light source detected
cyan	obstacle away	light source away
green	obstacle nearby	light source nearby
yellow	obstacle close	light source close
red	obstacle very close	light source very close

Table 2.2: Khepera sensor values

return) key after editing each coordinate or orientation field, so that your changes are taken into account.

Moving the robot this way won't prevent it from entering obstacles! However, when the robot moves on its own (i.e., with a controller), it cannot penetrate obstacles. The `Position` check box allows one to enable and disable the display of these `X`, `Z` and α coordinates. It might be useful to disable such a display in order to accelerate a little bit the simualtion.

2.6.4 Sensor representation

The robot properties window contains a lot of information on the robot sensors and motors. The Khepera robot has eight infra-red sensors all around allowing it to measure distance from obstacles by emitting infra-red light and measuring the quantity of reflected light. These sensors can also be used as passive sensors to measure the ambient infra-red light. Each of the infra-red sensors is represented as a rectangle for distance measurement and as a triangle for light measurement. These rectangles and triangles appear most often blue but their color actually ranges from blue to red according to the measurements performed by the corresponding sensor as explained in table 2.2.

Light sources are typically `Lamp` objects while obstacles can be either `Can`, `Box`, robots or `Wall` objects. Numerical values are also available for both distance measurement and light measurement. However, only one numerical value can be displayed at a time for each sensor. You can choose between the light and the distance numerical value with the corresponding check boxes situated at the left hand side of the window. Numerical values can be described as follows:

- Distance values range from 0 (no obstacle detected) to 1023 (obstacle very close).
- Light values range from 0 (light source very close) to 512 (no light source detected).

Since these values are noisy, just like with the real sensors, they are always oscillating, giving a rough approximation of the actual physical value.

The model of the infra-red sensors is based upon a lookup table for light and distance measurement since the response is non-linear with the distance. The color of object has some effect on

the distance measurements since red or white objects reflect more infra-red light than green or black objects. A random white noise of 10% is added to each measure (light and proximity) to make the modeling of the sensor realistic.

2.6.5 Motors representation

The Khepera robot has two independent motor wheels allowing it to move forward, to move backward, and to turn. Each of the two motors is symbolized by a square on the robot representation. This square contains an arrow indicating the velocity of the motor (direction and amplitude). A numerical value is also displayed which ranges from -20 to +20. Negative speed values can be used to make the robot go backwards. Each of these motor speed representations can be enabled or disabled by the `Motors` check boxes on the left hand side of the robot window. However, motor values can only be assigned by a robot controller, so you won't be able to observe different motor values unless you load a controller which is driving the motors and make it run (see later).

2.6.6 Jumper configuration

The jumpers represented below the left motor can be switched on and off by clicking on them. Since the robot can read these jumpers, this interface can be used as a communication way between the robot and you. However, the jumpers configuration should not be changed when using a real robot in remote control or download mode.

2.6.7 Extension turrets: K-Bus

Like on the real Khepera robot, it is possible with Webots to add extension turrets through the Khepera extension bus called the K-Bus. In order to plug-in an extension turret into your virtual Khepera, you just need to click on the `K-Bus` button. This will make a window appear in which you will be able to select an extension turret for the robot as depicted in figure 2.6.7.

The central popup menu in the bottom of the window will allow you to plug-in a number of extension turrets, including:

- `Khepera K213`: a black and white linear camera.
- `Khepera K6300`: a color video camera.
- `Khepera Gripper`: a gripper allowing Khepera to grasp `Can` objects.
- `Khepera Panoramic`: a 240 degree linear black and white panoramic camera.

More information on these extension turrets will be detailled later in this manual.

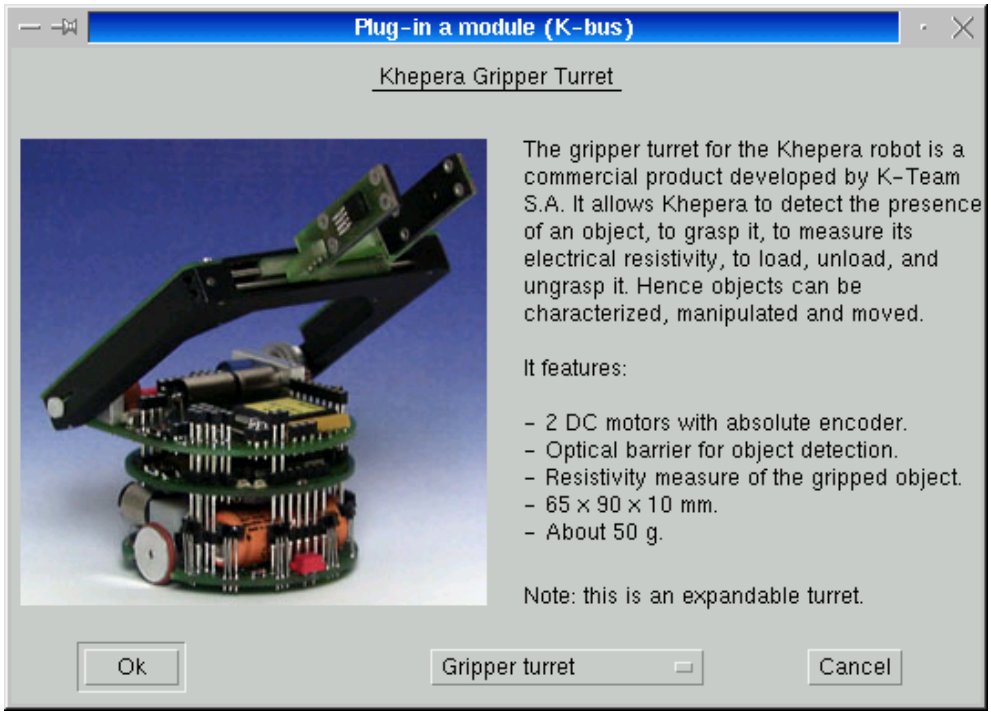
Plug-in a module (K-bus)
Khepera Gripper Turret
The gripper turret for the Khepera robot is a commercial product developed by K-Team S.A. It allows Khepera to detect the presence of an object, to grasp it, to measure its electrical resistivity, to load, unload, and ungrasp it. Hence objects can be characterized, manipulated and moved.
It features:
- 2 DC motors with absolute encoder.
- Optical barrier for object detection.
- Resistivity measure of the gripped object.
- 65 x 90 x 10 mm.
- About 50 g.
Note: this is an expandable turret.
Ok
Gripper turret
Cancel

Figure 2.13: Khepera bus plugin

2.6.8 Khepera controllers

In order to load a controller into the memory of a virtual robot, you will need to click on the `Controller` button. Then, a file selector window will allow you to look for a directory whose name ends up with the `.khepera` suffix. Such a directory is a controller directory for the Khepera robot. It contains a number of files which will be explained later in chapter 4. As an example, you can try out the `braiten.khepera` controller. This controller will make the robot move and avoid any obstacle.

A controller is actually a simple program written in C or C++ language. It can be compiled either by you or by Webots. Then, Webots launches the controller as an independent process and communicates with it through inter-process communications systems.

A GUI (Graphical User Interface) is also available and will allow you to set-up windows displaying your data and design your own user interfaces fulfilling your specific needs.

2.6.9 Moving to the real Khepera via the serial connection

From the robot properties window, it is very easy to switch from a virtual robot to a real robot. In order to proceed, you will need a real Khepera robot connected via a serial link to your computer. From Webots you can choose whether to control the real robot from the simulator or to cross-compile and download the controller to the real robot. In the first case, data is being sent

continuously from Webots to the Khepera and vice versa, i.e., to read sensor data and send motor commands. In the second case, the robot controller is cross-compiled for the real robot processor and downloaded to the real robot which in turn executes the controller on board and no longer interacts with the simulator.

2.6.9.1 Remote control of the real Khepera

Here are the steps to follow in order to set up a real robot connected to the simulator:

- Load a controller into the simulated robot (`braiten.khepera` is a good example for testing the real robot control).
- Connect the real robot to one the serial ports of your computer (see the manual of your robot to proceed).
- Set the robot serial mode to 38400 baud (mode 3) on both the simulated robot and the real robot. Hence, the jumper configuration of the real robot should correspond to the jumper configuration of the simulated robot. The jumpers of the simulated robot can be configurated from the widget situated just below the left motor in the robot window. However, the default configuration of the simulated robot is actually mode 3 which corresponds to 38400 bauds. Note that slower serial modes can be used as well, i.e., mode 1 and mode 2.
- Check that the serial ports are properly configured by checking the preferences window (`Options` menu, `Preferences...` item). The serial ports A and B should point to the corresponding directories. Default values should be all right. Normally these directories should be readable and writable from any user (`crw-rw-rw-`). If not, set the attributes of these files properly using the `chmod` command as `root`.
- In the upper left corner of the robot window, change the popup menu from `Simulated` to `Remote Serial A` or `Remote Serial B` according to the port on which the robot is connected.
- A popup window should inform you that everything goes right as depicted in figure 2.6.9.1.

- Click on the `Ok` button and run the simulation. The real robot should move while avoiding obstacles. You will be able to see the sensor values of the real robot in the robot window updated in real time.
- For timing reasons, the transfer to the real robot needs the speedometer indicates a value close to 1, which means the simulator runs close to real time. In order to achieve this, you should choose a time basis for the refresh (i.e., in 8, 16, 32, 64, 128 etc.) which give a ratio as close as possible to 1. The time basis value will depend on the complexity of scene, the power of your computer, the complexity of your control algorithm, etc.

Figure 2.14: Successful serial connection

2.6.9.2 Cross-compiling for the real Khepera

Here are the steps to follow in order to cross-compile and download the controller to the real robot:

- Load a controller into the simulated robot.
- Connect the real robot to one the serial ports of your computer (see the manual of your robot to proceed).
- Set the robot serial mode to 9600 bauds (mode 5) on both the simulated robot and the real robot. Hence, the jumper configuration of the real robot should correspond to the jumper configuration of the simulated robot. The jumpers of the simulated robot can be configured from the widget situated just below the left motor in the robot window.
- Check that the serial ports are properly configured by checking the preferences window (`Options` menu, `Preferences...` item). The serial ports A and B should point to the corresponding directories. Default values should be all right. Normally these directories should be readable and writable from any user (`crw-rw-rw-`). If not, set the attributes of these files properly using the `chmod` command as `root`.
- In the upper left corner of the robot window, change the popup menu from `Simulated` to `Download Serial A` or `Download Serial B` according to the port on which the robot is connected.
- If the controller has cross-compiled correctly, a popup window should inform you that the binary file is being transferred to the real robot (figure 2.6.9.2). After that, the controller will be automatically executed on the robot. If the robot has on-board batteries, you might want to unplug it from the serial connection.

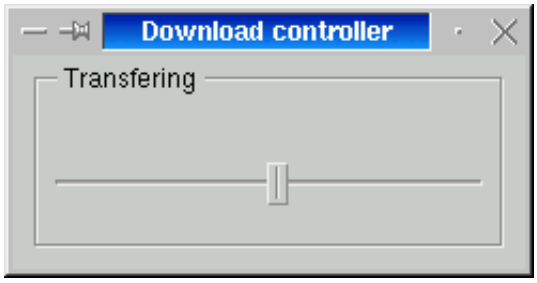

Figure 2.15: Downloading a cross-compiled controller

2.6.9.3 Software requirements

In order to make use of the cross-compilation feature of Webots 2.0 you must have properly installed the GNU C based cross-compiler for the Khepera robot in your system. Please refer to the corresponding software and manuals provided by K-Team to set up such an environment.

2.7 Working with Supervisors

A supervisor is a program similar to a robot controller. However, such a program is not associated to a robot but to a world. It can be used to supervise experiments. The supervisor program is able to get information on robots and on some objects in the world (like `Ball` objects) and to communicate with robots. A supervisor can be used to achieve the following:

- program inter-robot communications.
- record experimental data (like robot trajectories).
- control experimental parameters.
- display useful information from robots.
- define new abstract sensors.
- control the Alice robots.

Like with robot controllers, a GUI library is available to set up windows with buttons and other widgets. Such windows are useful to display and control what's going on in the experiment. Supervisor programming is described in chapter 5 of this manual.

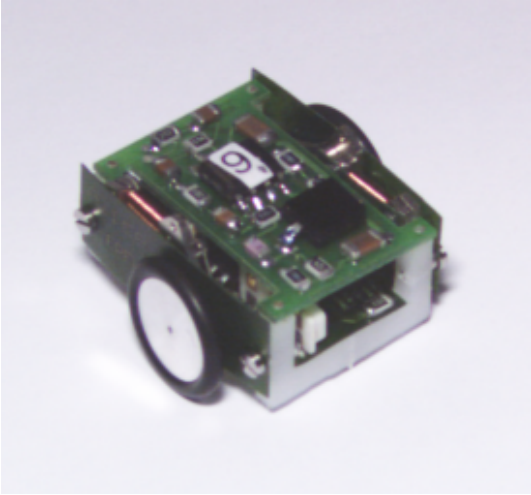

Figure 2.16: The Alice robot

2.8 Working with the Alice robot

2.8.1 Alice overview

The Alice robot is a very small and cheap autonomous mobile robot. It was developed by Gilles Caprari from EPFL - DMT - ISR - ASL. Its size (23mm x 21 mm x 16 mm) makes it suitable to study collective behavior with a large quantity of robots on a table. Two watch motors with wheels and tires can drive Alice at a speed of 20 mm per second. A PC16C84 microcontroller with 1Kword of EEPROM program memory allows developers to reconfigure its intelligence. This robot is powered by two watch batteries providing it with an autonomy of about 8 hours.

A number of extension turrets can be plugged-in on the top of the robot. The only extension turret used in Webots is the IRCom turret, which is depicted on figure 2.16.

2.8.2 Programming the Alice robot

In its current configuration, the Alice robot needs to be remote controlled, using an infra-red remote controller. In order to receive the infra-red signals from the remote controller, Alice needs an infra-red reception turret called Alice IRCom. Actually, this turret fulfill two functions: (1) it receives IR signals from the remote controller and (2) two other infra-red sensors can be used for obstacle avoidance, using the same principle as with the Khepera robot.

This infra-red remote control is modeled within Webots through the use of the Supervisor API. A supervisor controller can control up to 8 Alice robots in a world by sending them messages. Hence, there is no Alice controller programs, but only supervisor controllers sending messages to the Alice robots. Such messages indicate a pre-wired behavior to be executed by the Alice

robot. This technique, as well as the complete description of messages, is explained later in this manual.

2.9 Miscellaneous

2.9.1 Hiding and showing the buttons

The `Options` menu allows you to hide or show the two rows of buttons in the world window. The `Hide Edit Buttons` item will remove the buttons used for editing the world while the `Hide Simulation Buttons` item will remove the buttons controlling the simulation. When the buttons are hidden, the same menu will allow you to display the buttons back using the `Show Edit Buttons` or `Show Simulation Buttons` items.

2.9.2 Setting up preferences

The `Preferences...` item in the `Options` menu allows you to set-up the serial port device files as seen previously and also to set-up a number of default paths and files (see figure 2.9.2).

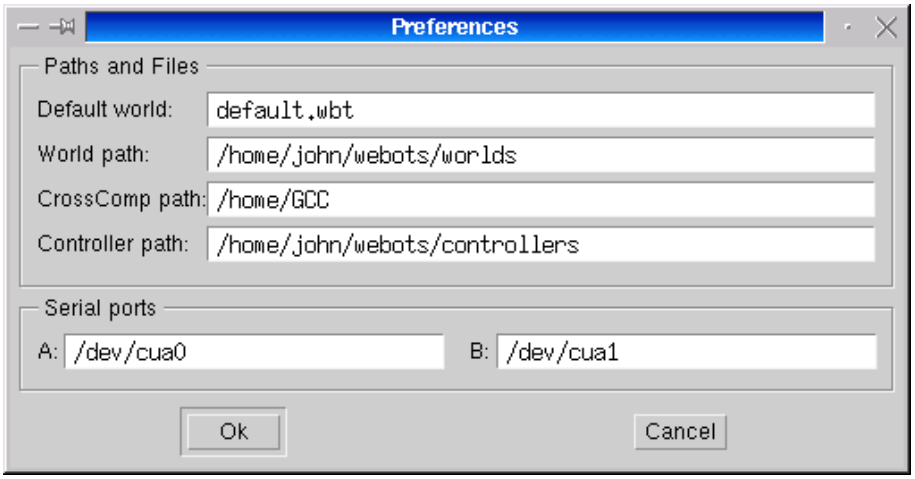

Figure 2.17: Webots preferences

The `Default world` is changed each time you quit Webots. It is replaced by the current world when Webots quits.

The `World path` is the path in which Webots is looking for world files. It is changed when successfully loading a world from a new directory.

The `Controller path` is the most important since it is the path in which Webots will look for controllers directories when loading a world containing a supervisor and / or robots. Note that Webots will also look in its own `WEBOTS_HOME/controllers` directory. Hence, if you

load a world and neither the controller path nor the `WEBOTS_HOME/controllers` directories contain the requested controllers mentioned in this world, then Webots will fail running this world. So, when setting up your own development environment, we recommand you that you create your own `controllers` directories for robots and supervisors in your home directory (or sub-directories) and you set up appropriately the corresponding paths in the Preferences window (although Webots should set it automatically when you load a new controller).

2.10 Directory structure

The `webots` directory typically located in `/usr/local` is organized as depicted in figure 2.10.

`webots/`	`LICENSE`	license text
	`controllers/`	controller program examples
	`images/`	images used internally by webots
	`include/`	include files necessary to the controllers
	`lib/`	libraries needed by the controllers
	`webots`	Webots application
	`webotsrc`	default .webotsrc file
	`webots.key`	license file
	`webots.xpm`	Webots logo
	`worlds/`	environment examples

Figure 2.18: Webots directory structure

Chapter 3

Sample Webots Applications

This chapter provides an overview of the sample applications provided within the Webots package. All these examples correspond to world files located in the `webots/worlds` directory and can be easily tested while reading this chapter. The source codes for the examples are located in the `webots/controllers` directory. For each example, a short description is provided. However, this description does no attempt to be exhaustive and should be considered as an introduction. More details can be found in the source code.

3.1 simple.wbt

This is the default world. It is a very simple square area with a few colorful boxes and a Khepera robot. The Khepera has a Braitenberg controller (source code in the `braiten.khepera` directory) and no extension turret, and therefore moves forward while avoiding obstacles.

3.2 five.wbt

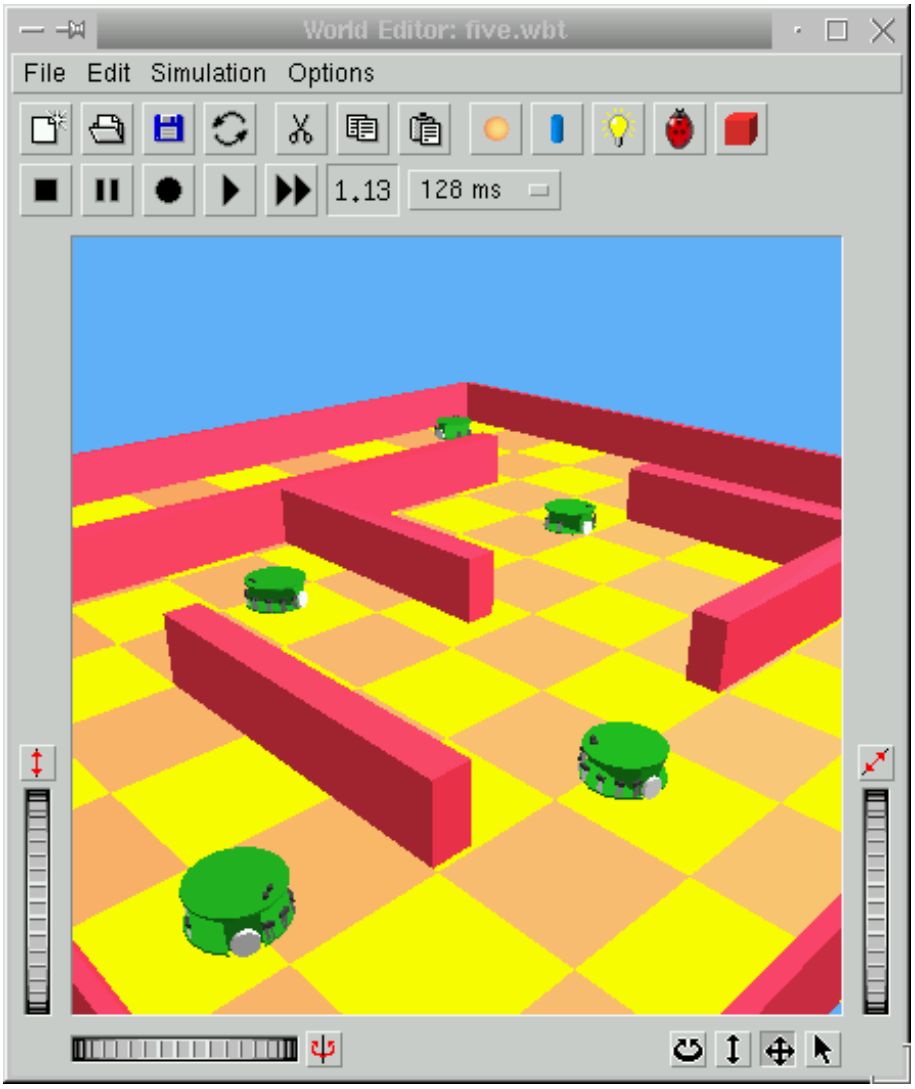

Figure 3.1: five.wbt

This example is a bit more interesting since five Khepera robots are moving in a maze. Each of these robots has a Braitenberg controller loaded. Running the simulation will make all the robots move in the maze while avoiding each other as well as obstacles. It can be entertaining to select one robot and choose the `Robot View` item in the `Simulation` menu, then to zoom on this robot using the control wheels for navigating in the scene. The source code for these controllers is in the `braiten.khepera` directory.

3.3 phototaxy.wbt

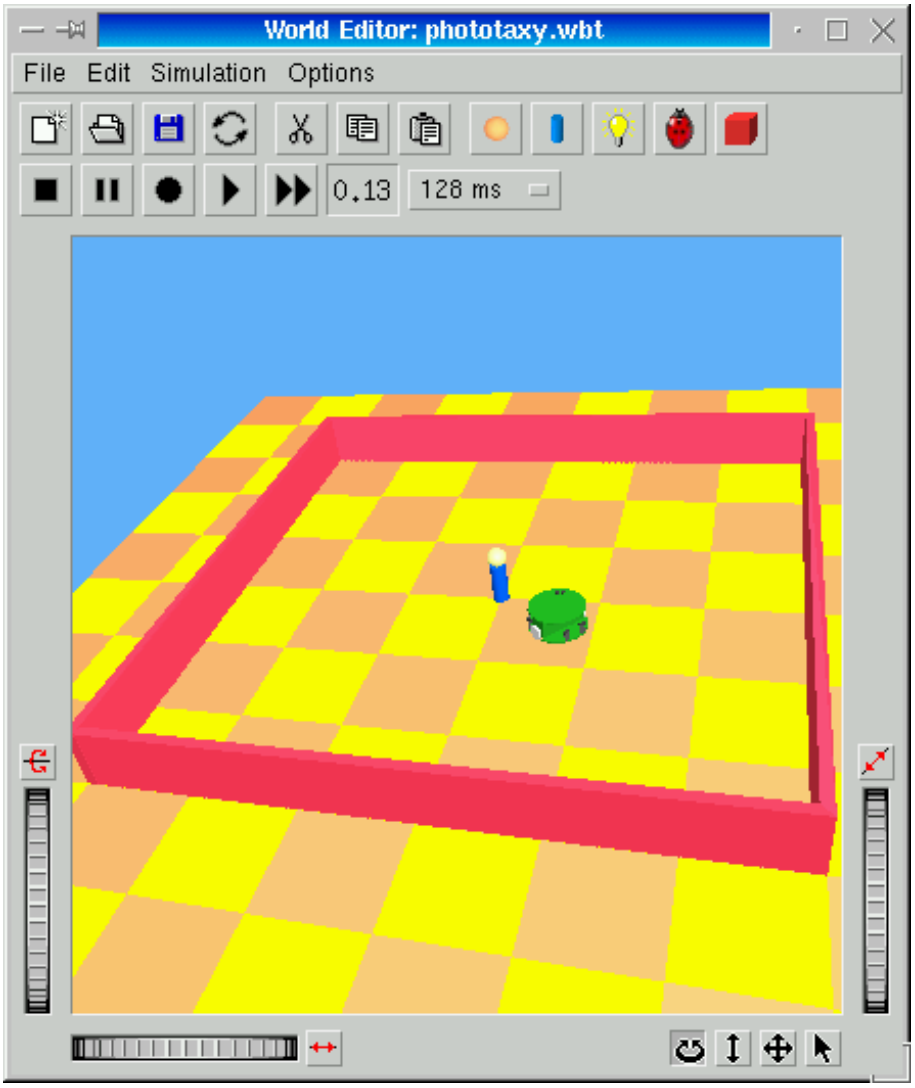

Figure 3.2: phototaxy.wbt

This example makes use of the light measurement capability of the infra-red sensors of the Khepera robot. A light source is set in the center of the environment and the robot uses light measurement information to try to find it. You will see the robot moving toward the light and avoiding the foot of the light (which is an obstacle) and going away from the light, then coming back and so on. The source code for this controller is in the `phototaxy.khepera` directory. It can be entertaining to drag the light source with the move cursor, to observe the robot running after the light.

3.4 jumper.wbt

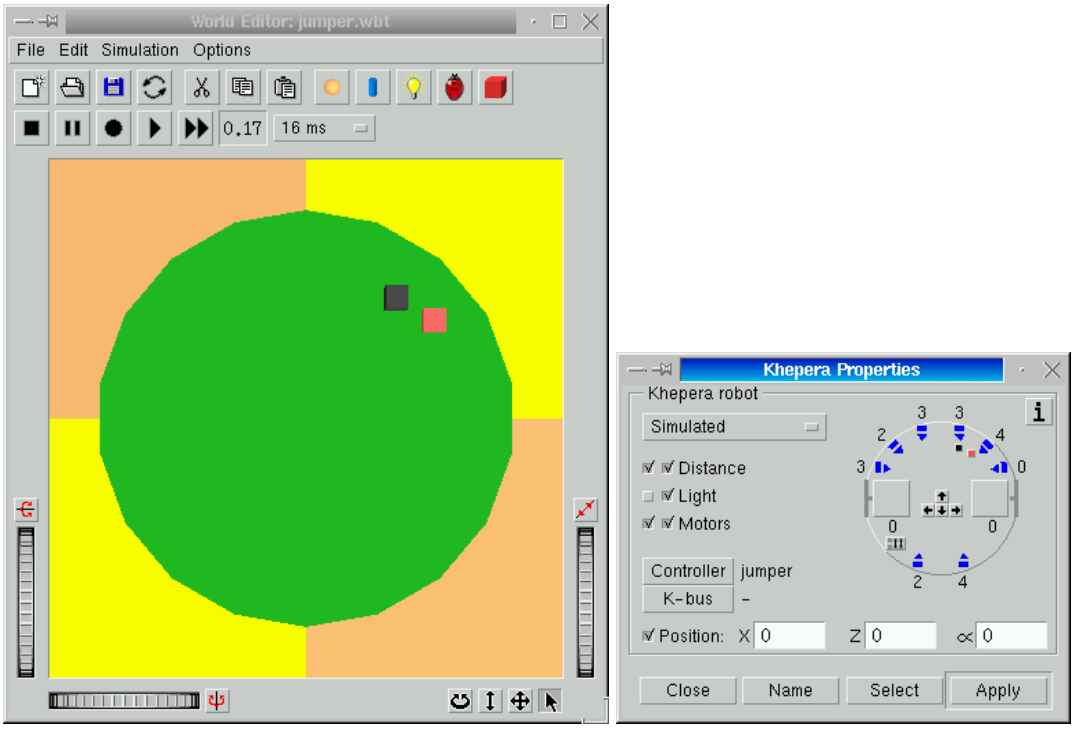

Figure 3.3: jumper.wbt

In this example, the robot is not moving. However, interesting things will happen. Run the simulation and double-click on the robot to have the robot window displayed. Then, click on the jumpers on the robot window (jumpers are just below the left motor of the robot). You will be able to observe that setting the left most jumper will cause the right most LED of the robot to be switched on. Unsetting this jumper will cause the same LED to be switched off. The middle jumper controls the other LED, while the last jumper has no effect. The source code for this controller is in the `jumper.khepera` directory.

3.5 finder.wbt

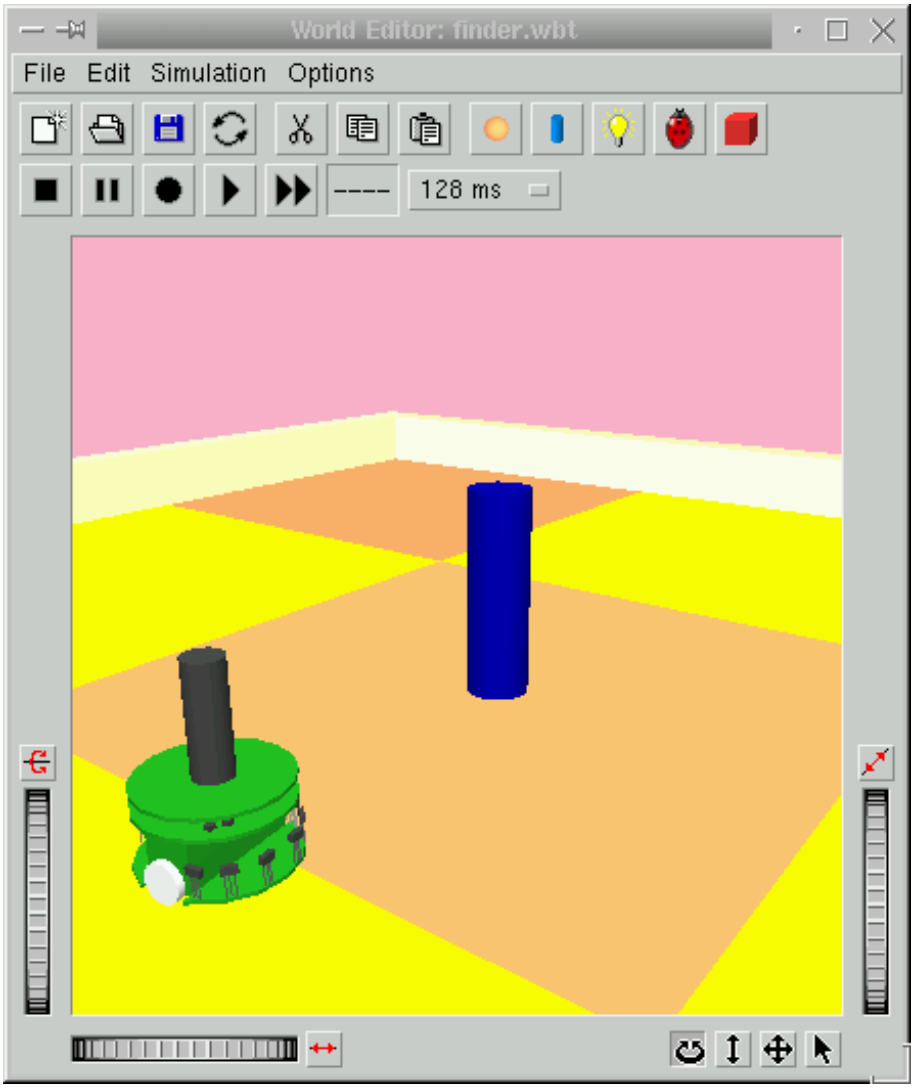

Figure 3.4: finder.wbt

This example shows how to use the Panoramic turret to reach a blue target cylinder situated in the center of the environment. The robot controller has two behaviors: (1) look for a dark object in the linear vision array and turn to set this object in the center of the field of view and (2) go forward and avoid any obstacle. As a result you will be able to observe the robot going very efficiently towards the blue cylinder and then turning around the cylinder. This can be explained by the fact that the cylinder is also an obstacle and the obstacle avoidance behavior has a strongest priority over the goal seeking behavior. The source code for this controller is in the `finder.khepera` directory.

3.6 can.wbt

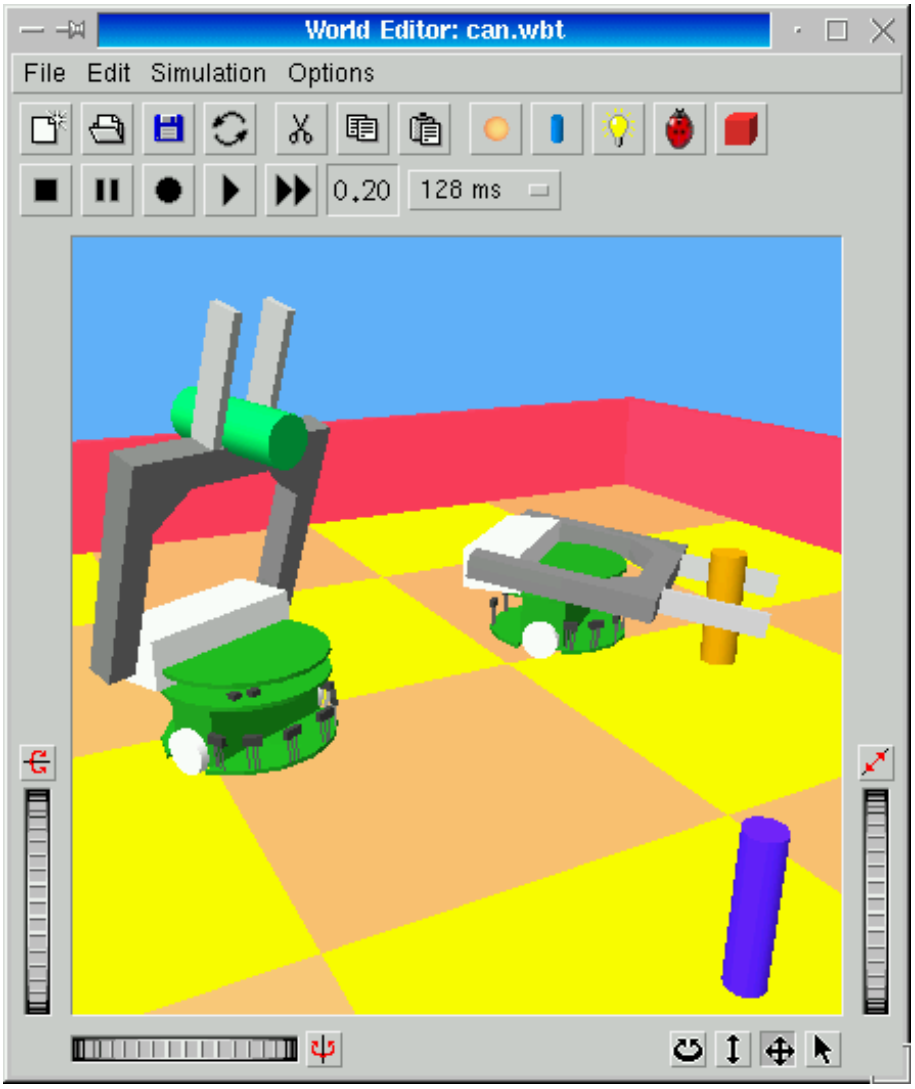

Figure 3.5: can.wbt

This example demonstrates two Khepera robots equipped with Gripper turrets and running the same controller program. The robots look for small objects which can be grasped by the gripper. Such objects are identified through the use of infra-red sensor distance measurements. If only one or two sensors in front of the robot are sufficiently excited and no other sensor is excited, then the robot deduces it faces a small object, which can be grasped by the gripper. Then, it moves the gripper down, grasps the object and moves the gripper up. After that, the robot continues moving, looking for another object to grasp. When it finds one, it puts down the object it held on its side and grasps the newly found object. And this behavior is repeated for ever... The source code for these controllers is in the `can.khepera` directory.

3.7 attacker.wbt

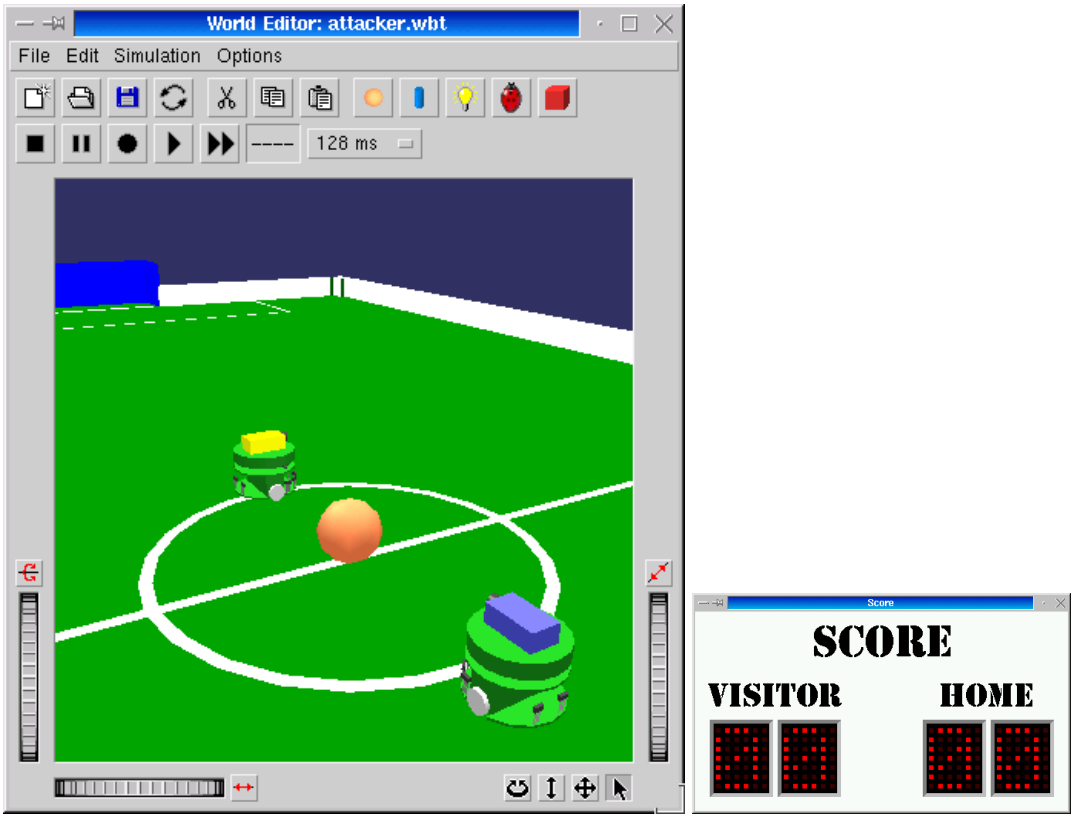

Figure 3.6: attacker.wbt

The Khepera robots are trying to score a goal on a soccer field! In order to achieve this, the robots use some image processing from the K6300 turret to localize the ball and the goal. Then, the robots move to the right position to kick the ball in order to get it closer to goal. The robots will repeat this behavior 3 or 4 times before a goal is scored. However, it sometimes happens that the robots get frozen for some mysterious reasons (probably due to the complexity of the algorithm). In this case, we recommend that you move the robots a little bit (with the keyboard arrows) and / or then reload the corresponding controllers to restart them. The source codes for these controllers are in the `attacker_blue.khepera` and `attacker_yellow.khepera` directories.

3.8 buffer.wbt

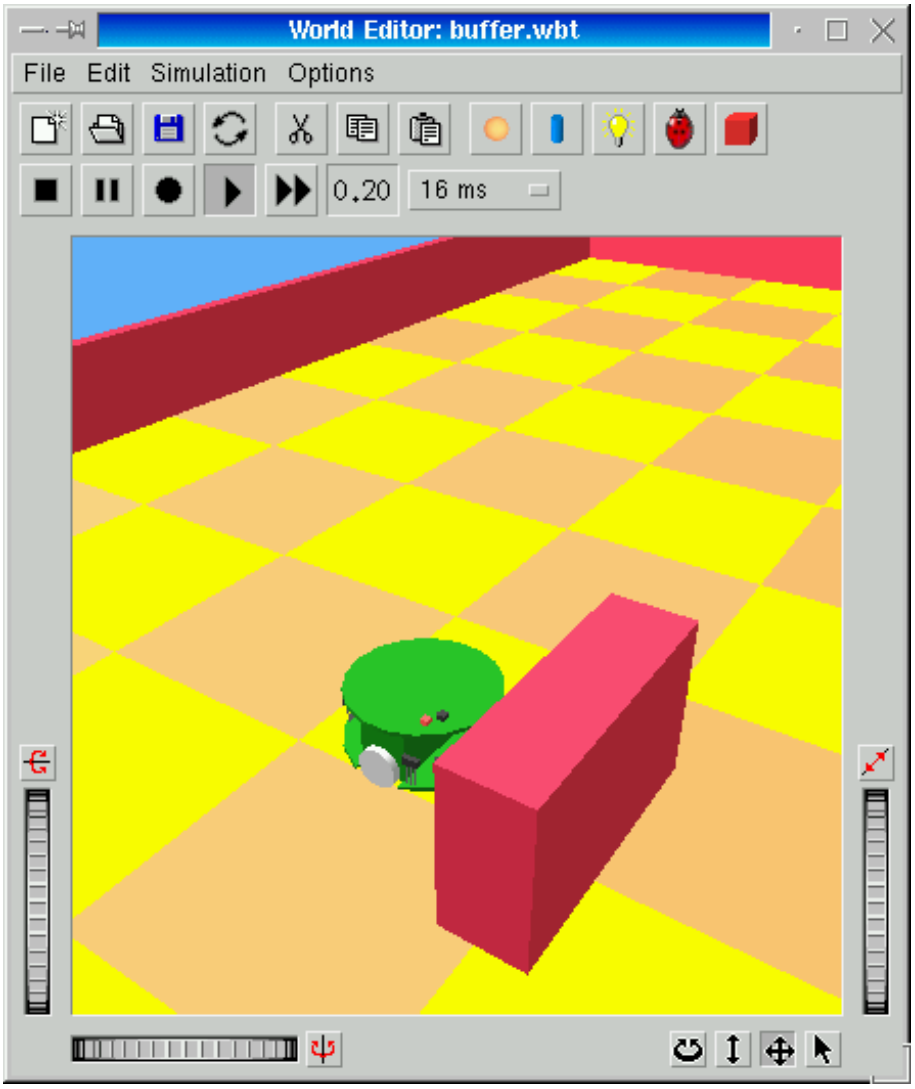

Figure 3.7: buffer.wbt

This example demonstrates how it is possible to exchange information between a supervisor and a robot. The robot is communcating with the supervisor to say when it feels an obstacle in front of it. Then, the supervisor decides that if the object is close enough to the robot, the robot should turn on one LED. The supervisor, then, sends a message to the robot meaning to turn on (or off) the LED. Similar methods can be used to implement some communication between robots via the supervisor. The source code for the supervisor is in the `buffer.supervisor` directory and the source code for the robot controller is in the `buffer.khepera` directory.

3.9 town.wbt

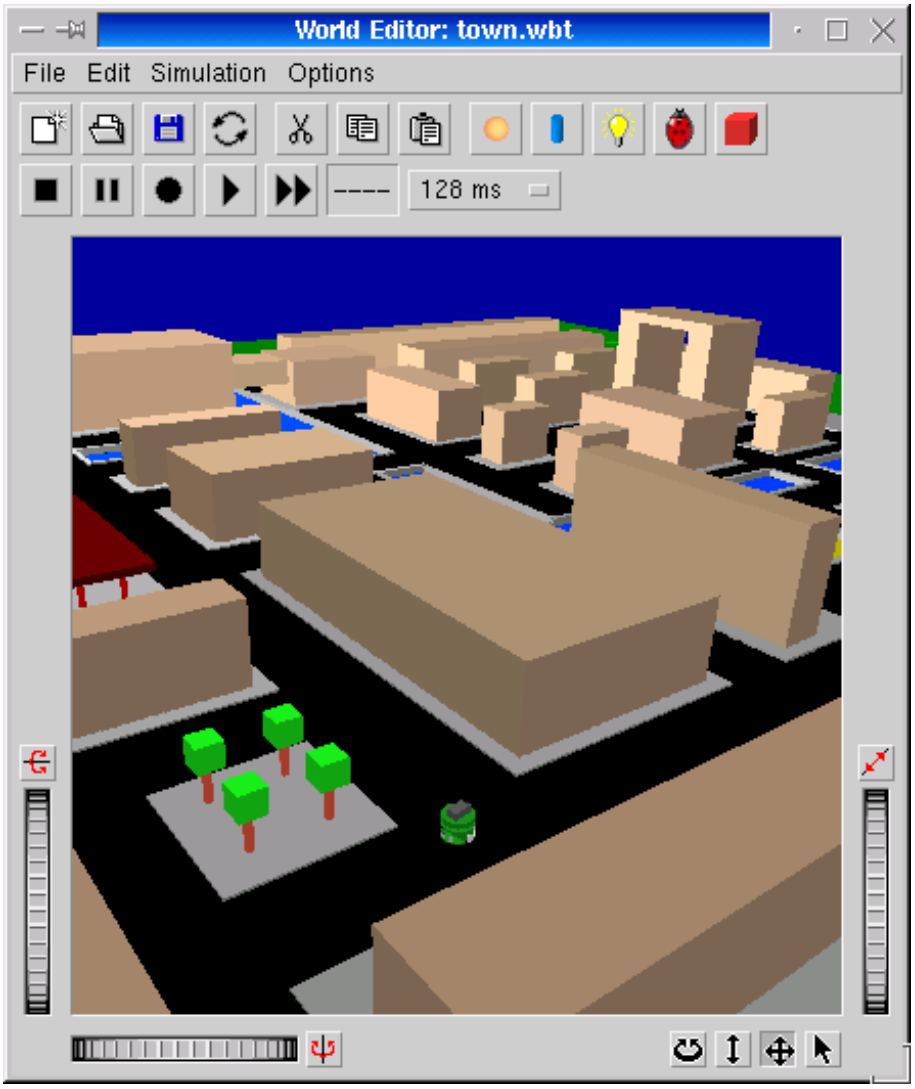

Figure 3.8: town.wbt

This example was developed by Grald Foliot (Lyon 2 University, France). It models a city sized for the Khepera robot with buildings, streets, rivers, parks, trees, etc. in which you can run your own experiments. Such an environment is well suited for navigation tasks because a lot of landmarks can be recognized by the robots equipped with vision sensors. Note that this world makes an extensive use of VRML nodes which were not created with Webots, but with a text editor.

3.10 house.wbt

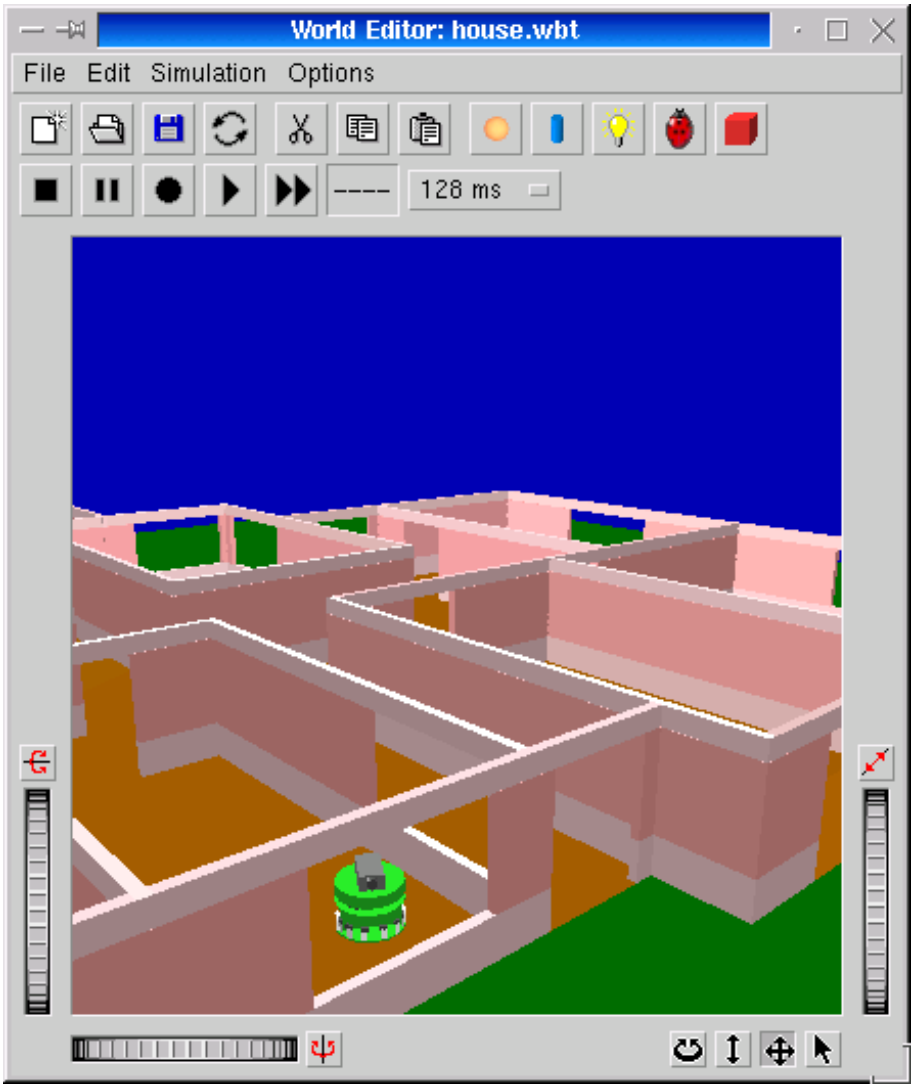

Figure 3.9: house.wbt

This example, also developed by Grald Foliot, models a simple schematic house sized for the Khepera robot with doors, windows, corridors and rooms in which you can run your own experiments. Such an environment is well suited for indoor navigation tasks for which the problem of recognizing and passing doors is an important issue. The robots is equipped with a vision sensor since it seems to be the most suitable type of sensors for such environments. Note that this world makes an extensive use of VRML nodes which were not created with Webots, but with a text editor.

3.11 chase.wbt

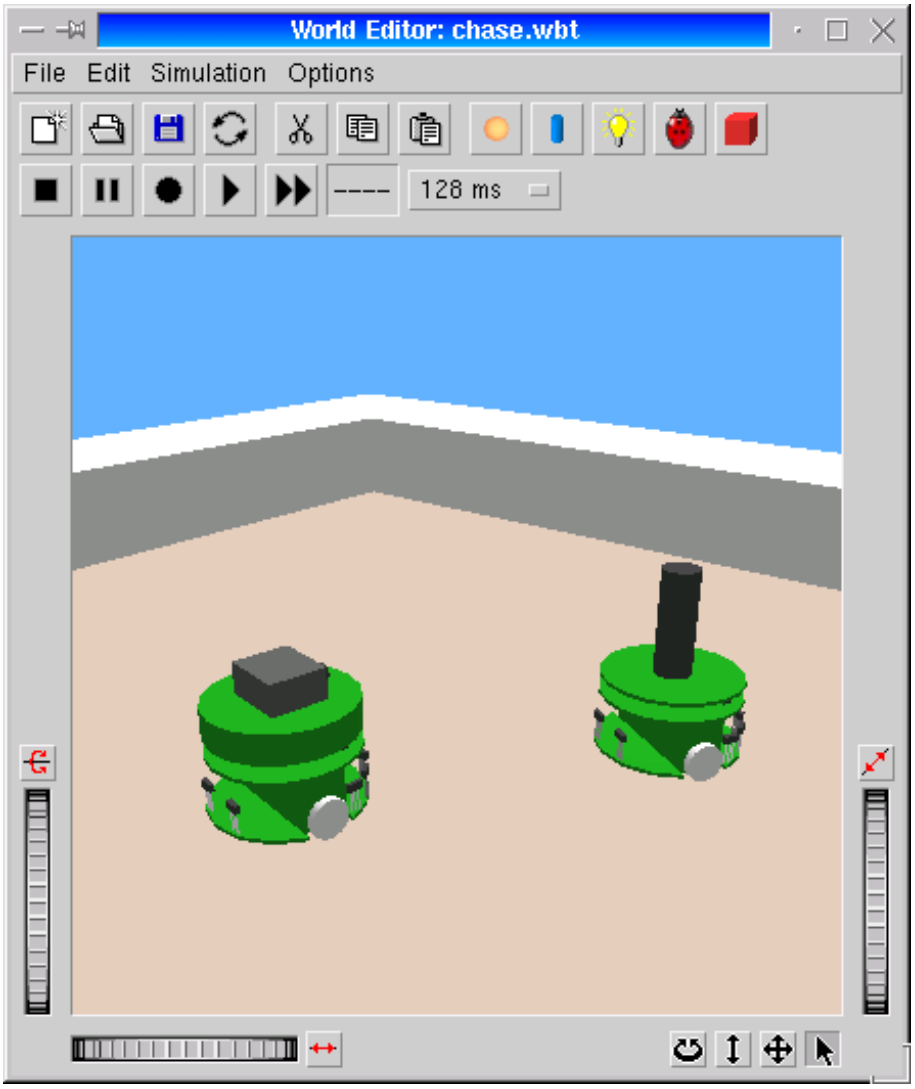

Figure 3.10: chase.wbt

This experiment shows a record of the genetic evolution of neural networks for the prey-predator experiment developed by Dario Floreano (LAMI - EPFL, Switzerland). The robot with the K213 vision turret is the predator and is trying to catch the prey while the prey relies only on infra-red sensors to escape the predator. Note that the prey is equipped with a Panoramic turret but it doesn't use it at all. It stands just here because of its shape and color to help the predator seeing the prey! This experiment is a good example of communication between the supervisor and the robots. The source codes for this experiment are in the `chase.supervisor`, `predator.khepera` and `prey.khepera` directories.

3.12 stick_pulling.wbt

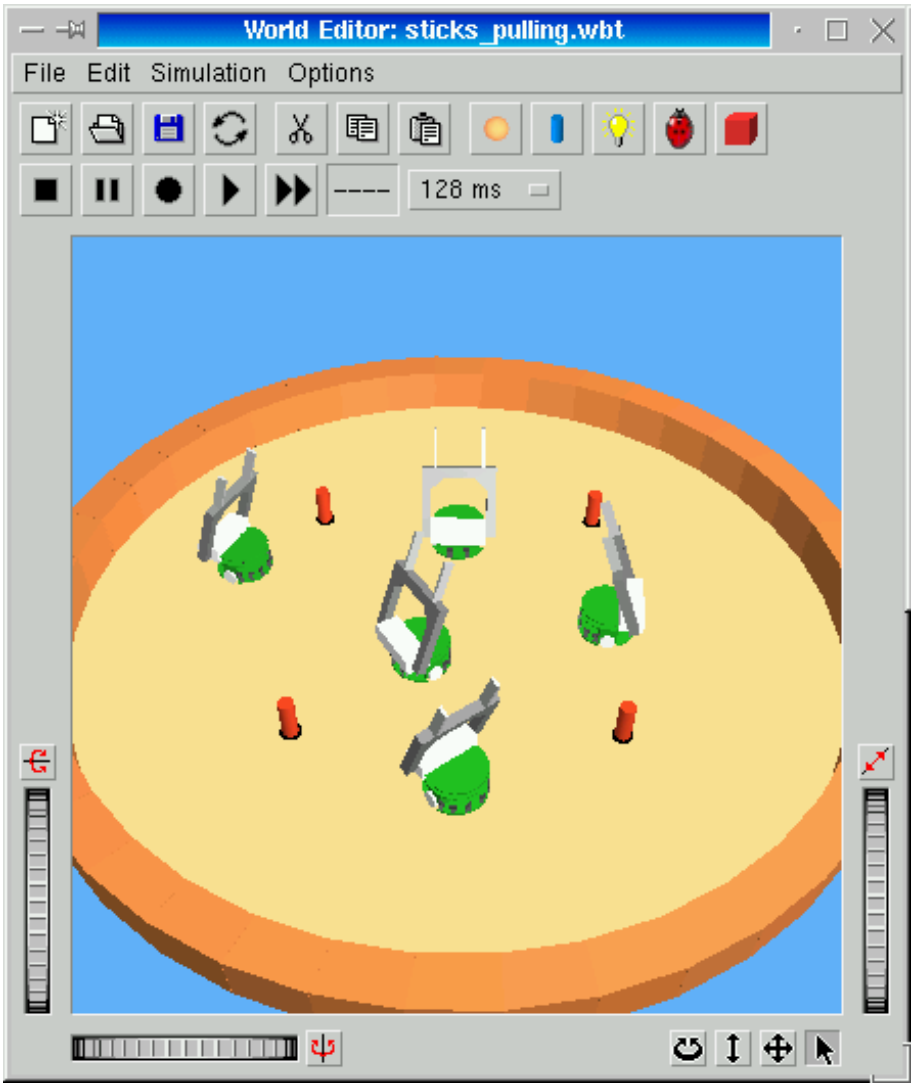

Figure 3.11: stick_pulling.wbt

This experiment of collective robotics was developed by Auke-Jan Ijspeert (IDSIA, LAMI-EPFL, Switzerland). Five robots collaborate to extract wood sticks from the ground. These sticks are too long to be extracted by a single robot equipped with a gripper, hence, the robots find, grasp, pull the stick and wait for help from another robot. The source codes for this experiment are in the `stick_pulling.supervisor` and `stick_pulling.khepera` directories.

3.13 alice.wbt

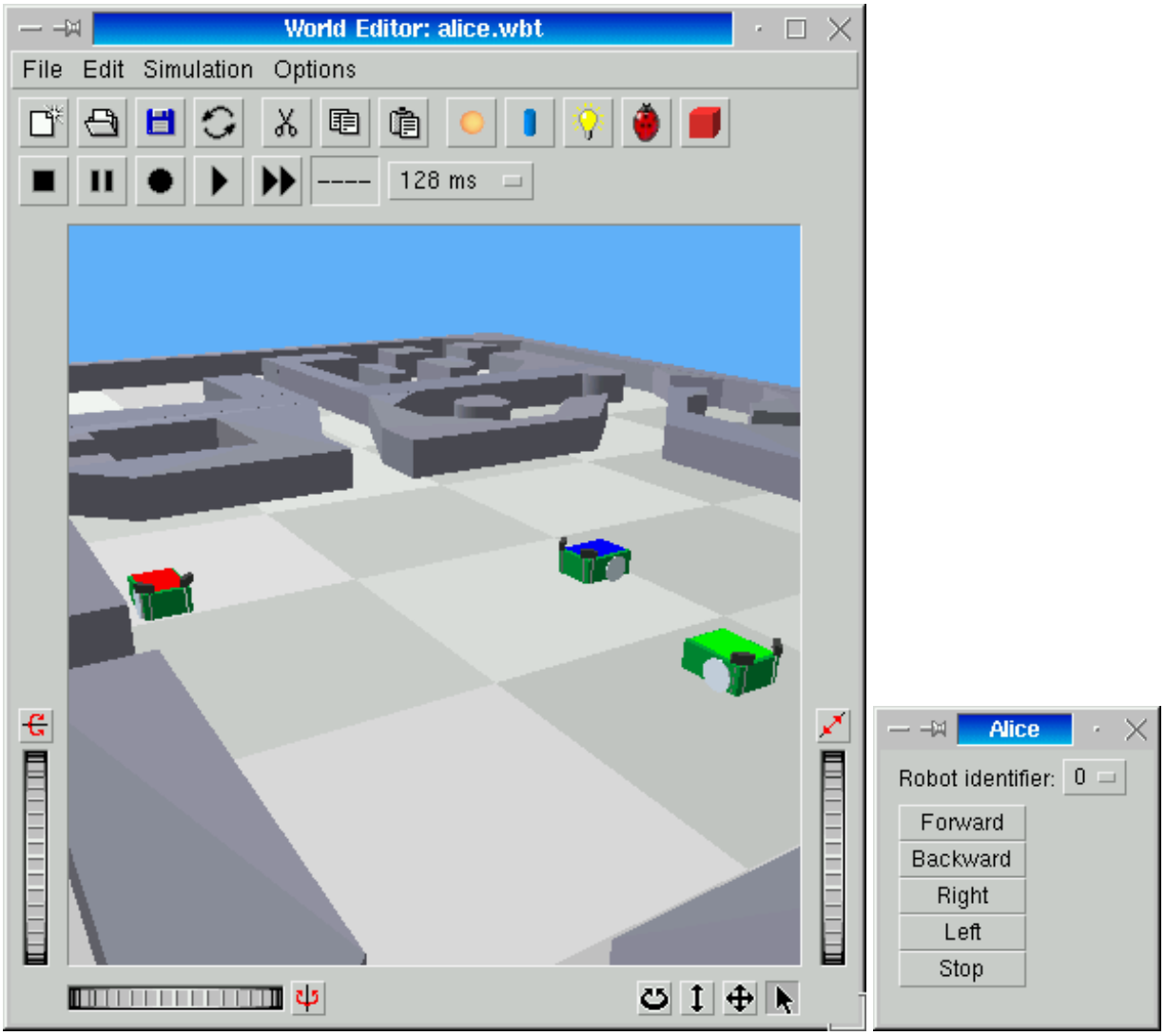

Figure 3.12: alice.wbt

This example shows how to control several Alice robots from a supervisor by sending messages to the robots. A graphical user interface allows the user to select an Alice robot though a popup menu and send it some command. The source code for the supervisor controller is in the `alice.supervisor` directory.

Chapter 4

Programming with the Khepera API

4.1 My first Khepera program

4.1.1 Source code

Here is one of the simplest C program you can write to control the Khepera robot:

```
#include <Khepera.h> /* We just need Khepera stuff */
void main()
{
  khepera_live(); /* Initializes the robot */
  for(;;) /* Endless loop */
  {
    khepera_set_speed(LEFT,5);  /* Set the motors to */
    khepera_set_speed(RIGHT,5); /* the same speed, */
    khepera_step(64); /* run one step of 64 ms */
  } /* and repeat this forever... */
}
```

This very simple program will make the Khepera robot move forward until it bumps into an obstacle, if ever. The speed of the Khepera will be of 40 mm/s since the speed unit is 8 mm/s for the `khepera_set_speed` function. If you set different speeds to each motor, you will observe the robot follow a circular trajectory. You may also set negative speed values and observe the robot going backwards. Finally, if you set opposite values to the left motor and to the right motor, you will be able to see the robot spinning on itself like a top.

This program, however, illustrates some fundamental aspects of programming a robot within Webots. As you can see, such a program never ends. It first performs some initialization and enters an endless loop. So, it is not possible to exit the program from itself.

> Warning: Never invoke the `exit()` C function in a controller: this would freeze things up in the simulator.

4.1.2 Controller directory structure

The source code listed above is indeed the contents of a C source file named `crazy.c` which lies in the `crazy.khepera` directory (in the `controllers` directory). This name was chosen because the behavior of such a robot is somehow "crazy". Each controller directory's name ends up with a `.khepera` suffix. Other examples of `.khepera` directories are provided in the `webots/controllers` directory. Let's observe what is needed in a controller directory by listing the `crazy.khepera` directory:

```
ls crazy.khepera
Makefile    crazy.c    crazy.o    crazy
```

The `Makefile` is used to define the compilation procedure when the `make` command is used (type `man make`, for more information on `make`).

The `crazy.c` file is the C source code for the controller we have just seen above.

The `crazy.o` file is the object file generated by the compilation of `crazy.c`.

The `crazy` file is the executable file that is launched by Webots.

4.1.3 Compiling the controller

This controller can be compiled from the shell by issuing the following command:

```
make
```

However, you won't be able to compile this controller unless you are logged in as `root`, because you are not allowed to write in the webots directory. To compile it, you will first have to get a local copy of this controller as explained below.

4.1.4 Modifying the controller

To test the whole editing / compilation / run process, you can copy the whole `crazy.khepera` directory in your home directory:

```
cd ~mkdir webots cd webots mkdir controllers cd
controllers cp -r
/usr/local/webots/controllers/crazy.khepera .
```

and try to modify the `crazy.c` file (by changing the value of the right speed to -5 instead of 5, for example). Save the modified file and run the `make` command. If no compilation errors occur, new object and executable files should be created. You can now launch Webots and load your own `crazy.khepera` as a controller for a robot (be careful to select your own directory you have just created and not the default one which is usually `/usr/local/webots/controllers`).

From there, you can again modify the `crazy.c` source file. You can also add header files or new source files and modify the `Makefile` according to the source files which need to be compiled. It's much easier to build your own system starting from such an example rather than starting from scratch.

Note that the prefix of the name of a controller directory (i.e., `my_controller.khepera`) must be the same as the name of the executable file (i.e., `my_controller`). If this is not true, Webots will be unable to launch your controller.

4.2 Webots execution scheme

4.2.1 Khepera controllers

Indeed, the controller programs are under the control of the Webots simulator. In other words, they can be launched and quitted only by the simulator. The simulator operates during the `khepera_step` function. During this function call, it computes the requested sensor values and updates the position of the robot according to the motor values. Then it can either:

- return to the execution of the controller (`Run` and `Fast` modes).
- suspend the execution of the controller (`Step` and `Stop` modes).
- exit from and destroy the controller (when loading another controller, cutting the robot, or quitting Webots).

The `khepera_step` function simulates a number of milliseconds of a real robot running. Requested sensors have to be enabled before calling `khepera_step` and their values will be available after this call. Actuator commands have to be requested before calling to `khepera_step`, so that they are actually performed during this call. The figure 4.1 summarizes the structure of a standard controller program.

An explanation for the MODULE, SENSOR, and ACTUATOR concepts is given later in this chapter.

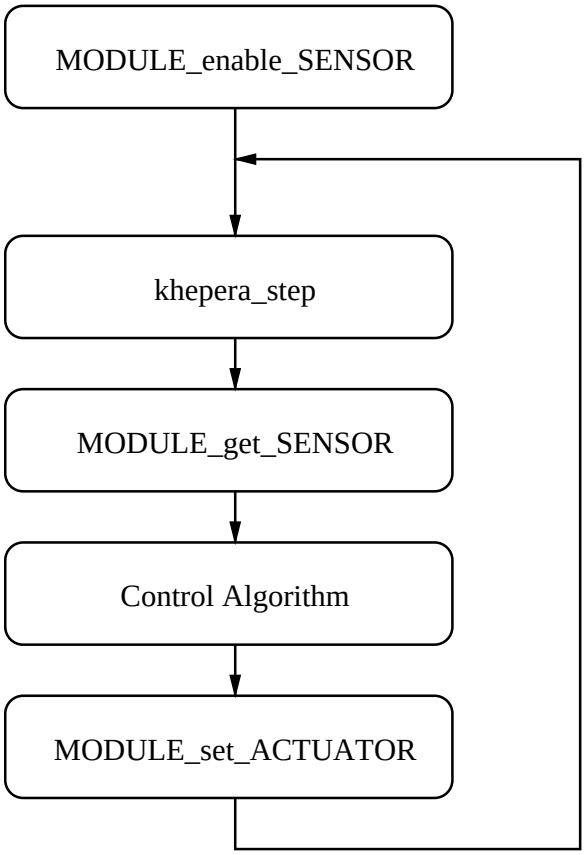

Figure 4.1: Structure of a controller program

4.2.2 Other controllers

The `webots` program successively calls the robots controllers and the supervisor program, if any. Here is the sequential order of such operations which are executed once during the `Step` mode and continuously during the `Run` and `Fast` modes.

- Execute the `khepera_step` or `supervisor_step` function for each controller (this includes the simulation of actuators and sensors).
- Simulate other things within Webots simulation (like the `Ball` movement).
- Update the display of the world (if not in `Fast` mode).

4.3 Getting sensor information

For each sensor you need, you will have to declare that you request the computation of the simulated sensor by a function call looking like `MODULE_enable_SENSOR`, where `MODULE` can be replaced either by `khepera`, `k213`, `k6300`, `gripper`, or `panoramic` and `SENSOR` can be replaced either by one of `jumper`, `position`, `proximity`, `light`, `speed`, `arm`, `grip`, `presence`, `resistivity`, `raw`, etc. By default, no sensor computation is performed, so it is important to request explicitly the sensors you need. You can also disable some sensors if you don't need them any more using the corresponding `MODULE_disable_SENSOR` function.

The following sample program implements a simple Braitenberg vehicle[1]:

```
#include <Khepera.h>
#include "braiten.h"

/* Interconnection matrix between IR proximity
   sensors and motors */
int32 Interconn[16] =
{-5,-15,-18,6,4,4,3,5,4,4,6,-18,-15,-5,5,3};

main()
{
  int32 i,left_speed,right_speed;

  khepera_live();
  khepera_enable_proximity(IR_ALL_SENSORS);
  for(;;) /* Khepera never dies! */
  {
    left_speed  = 0;
    right_speed = 0;
    for(i=0; i<8; i++)
    {                    /* Connections sensors-motors */
      right_speed += khepera_get_proximity(i)*Interconn[i];
      left_speed += khepera_get_proximity(i)*Interconn[8+i];
    }
    left_speed  /= 400;                  /* Global gain */
    right_speed /= 400;
    left_speed  += FORWARD_SPEED;        /* Offset */
    right_speed += FORWARD_SPEED;
    khepera_set_speed(LEFT,left_speed);   /* Set the motor */
    khepera_set_speed(RIGHT,right_speed); /* speeds */
    khepera_step(64); /* Run one step of 64 ms */
  }
}
```

[1] Braitenberg V., "Vehicles: Experiments in Synthetic Psychology", MIT Press, 1984.

This controller introduces infra-red sensor proximity readings which are used to drive the robot wheels so that the robot can avoids obstacles on its way. It is provided as an example controller in the `braiten.khepera` directory. The `int32` type corresponds to a signed integer on 32 bits. Similar other useful types are defined in the `types.h` include file. They should always be used to avoid platform dependent problems occuring when porting your software to another system.

Sensor values can be read by using the following generic `MODULE_get_SENSOR` function (see Reference Manual for details).

Here is the list of all the sensors available with the basic Khepera robot:

- `proximity`: 8 infra-red sensors used to measure the distance from the obstacles.
- `light`: the same 8 infra-red sensors used to measure the level of ambient light.
- `position`: value of the incremental encoder on each wheel (useful for odometry purposes).
- `speed`: velocity of each motor wheel.
- `jumper`: configuration of the jumpers on the upper side of the robot.

4.4 Controlling Actuators

Unlike sensors, actuators don't need to be enabled or disabled. They are always available. The generic `MODULE_set_ACTUATOR` functions are used to send values to actuators. The actuators of the basic Khepera robot include:

- `led`: the two LEDs which stand on the upper side of the robot.
- `position`: the internal value of each of the two incremental wheel encoders.
- `speed`: velocity of each of the motor wheels.

4.5 Working with extension turrets

Extension turrets can be plugged into the K-Bus of the robot as explained in chapter 2. In order to use extension turrets in controllers, you will have to include the corresponding include file (`KheperaK213.h`, `KheperaK6300.h`, `KheperaGripper.h` or `KheperaPanoramic.h`). Then, you will be able to use a number of functions to read sensor information or write motor commands which are specific to the turret you declared. The syntax and usage of each of these functions is detailed in the Reference Manual.

4.5.1 K213 turret

This turret features a 1D black and white linear camera device (see figures 4.5.4 and 4.3). It is commercially available from usual Khepera distributors.

This turret is useful for equipping the Khepera of a simple visual perception system. Many application can take advantage of such a system including, but not limited to, landmark recognition (natural optical pattern or bar code), optical flow, light source identification and visual stimuli classification.

The image produced by this camera is a 64 x 1 pixel image in 256 grey levels corresponding to a front view of 36 degrees (see figure 4.4). The pixel intensity is optimized to improve the contrast.

4.5.2 K6300 turret

This turret helds a 2D color camera digital device (see figures 4.5 and 4.6). Image processing can be performed by the robot itself.

The real turret corresponding to the K6300 turret in Webots will be available commercially for the Khepera robot soon. The black and white version will be called K5300 while the color version will be called K6300. However, since the real turret is still a prototype at the time, the model included in Webots may differ from the real device.

The `k6300_get_red`, `k6300_get_green` and `k6300_get_blue` functions are used to get RGB values of the image captured by the camera. These functions are implemented as macros or inline functions and hence are very efficient. However, you might want to use another function to get a pointer to an array containing the same data. The `k6300_get_image` function returns such a pointer, which might be more convenient, depending on the way you want to read the data.

In order to allow robots to recognize each other, for soccer games for example, the color of the K6300 turret can be defined using the RGB text fields in the robot window depicted in figure 4.7. In order to get faster simulations, the display of the image can be disabled by using the image check box.

4.5.3 Gripper turret

The gripper turret (see figures 4.8, 4.9 and 4.10) allows the robot grasp `Can` objects. Such objects can be transported to another location and put down on the floor by the robot. The `can.khepera` controller is a good example to start programming with the gripper.

Two motor commands are available on the gripper. The first one controls the position of the arm which can be set down, up, or in an intermediate position using the `gripper_set_arm` function. The position of the arm is an integer value which ranges from 0 to 255, but the useful range is [160,255]. A value of 255 means that the gripper is down in front of the robot while

a value of about 160 means that the gripper is up. The second motor command controls the position of the grip itself, that is, the two fingers of the device. You can close or open this device with the `gripper_set_grip` function, but no intermediate position is available.

However, when you close the gripper on an object, the object will stop the run of the grip at a given position. The position is indeed the size of the object and can be measured using the `gripper_get_grip` function.

The gripper turret also features two interesting sensors, which are an optical barrier and a resistivity measurement device. The first allows Khepera to detect if an object is inside the grip before or after the grip is closed by using the `gripper_get_presence` function. The second allows Khepera to measure the electrical resistivity of a gripped object by using the `gripper_get_resistivity` function.

The `K-Bus` button is reserved for future use. It is currently not possible to plug-in an additional turret over the gripper turret. Such a feature will be introduced in further versions.

The gripper can be controlled from the keyboard like the robot. You will have to press the SHIFT key simultaneously with the arrow keys of your keyboard to move the gripper up or down. You can also type a new value for the gripper arm position or use the little arrow buttons close to the `arm` text field. The space bar, or the `Gripper` button allows to open or close the gripper. However, `Can` objects cannot be grasped this way. Grasping of `Can` objects may occur only when the robot controller is running (`Run`, `Step` or `Fast` modes).

4.5.4 Panoramic turret

The real panoramic turret is actually not commercially avaible for the Khepera robot (see figures 4.11, 4.12 and 4.13). It is currently a prototype developed at LAMI - EPFL by Yuri Lopez de Meneses. This turret provides a black and white linear panoramic view around the robot. It cover 240 degrees on a linear array of 150 pixels. Each pixel has a resolution of 64 grey levels.

The sensors on this turret correspond to the different types of data you can get from the artificial retina. They include `raw` image data and some filtered image data (see the reference manual for more info on these filters). The raw data can be obtained from the `panoramic_get_raw` function. The `panoramic_set_parameters` function is used to set a number of parameters for filtered images.

The `panoramic_set_window` function allows one to optimize the speed of sensor computation by reducing the angle of view of the camera. It defines a window within the 150 pixel array that will be computed instead of computing each of the 150 pixels.

Finally, the `panoramic_set_precision` function can be used to reduce the resolution of each grey level pixel. This precision is expressed in bits and ranges from 1 to 6 bits (i.e., from black and white up to 64 grey levels).

The grey levels can be represented as a graph and / or as actual grey levels. Each representation can be disabled through the corresponding check boxes to achieve faster simulation.

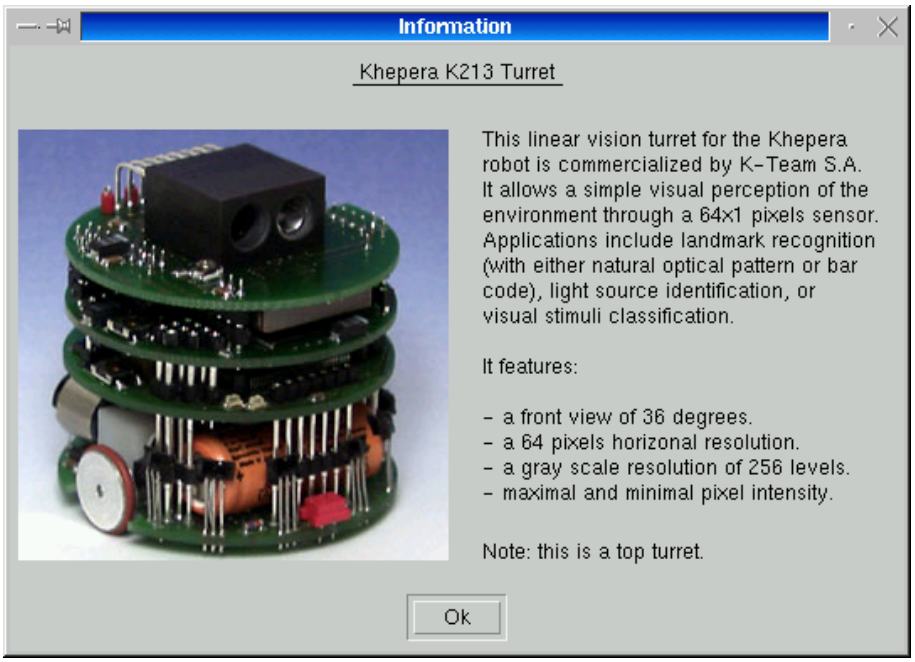
Information
Khepera K213 Turret
This linear vision turret for the Khepera robot is commercialized by K-Team S.A. It allows a simple visual perception of the environment through a 64x1 pixels sensor. Applications include landmark recognition (with either natural optical pattern or bar code), light source identification, or visual stimuli classification.
It features:
- a front view of 36 degrees.
- a 64 pixels horizonal resolution.
- a gray scale resolution of 256 levels.
- maximal and minimal pixel intensity.
Note: this is a top turret.
Ok

Figure 4.2: The real K213 turret

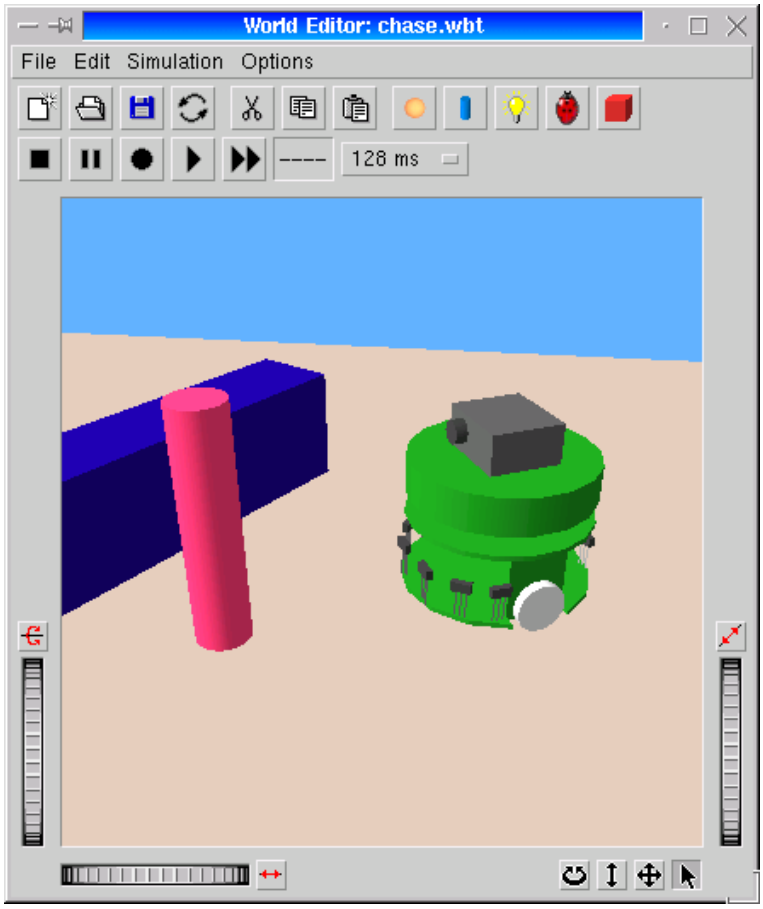

Figure 4.3: The simulated K213 turret

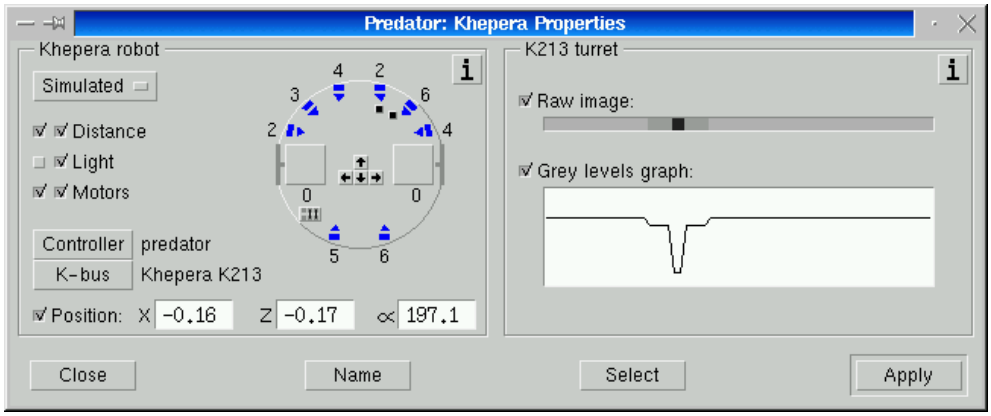

Figure 4.4: The robot window for the K213 turret

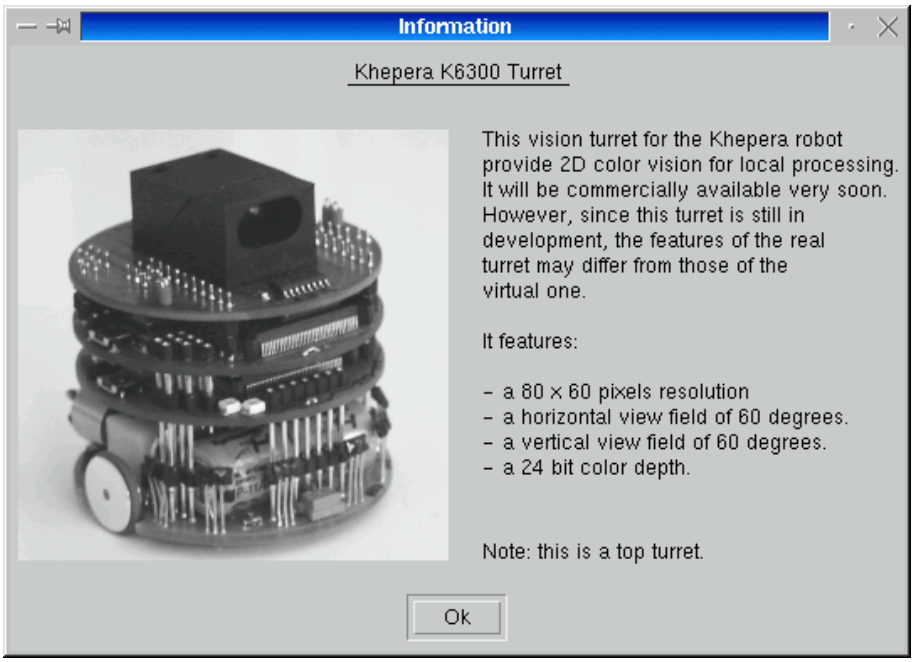
Information
Khepera K6300 Turret
This vision turret for the Khepera robot provide 2D color vision for local processing. It will be commercially available very soon. However, since this turret is still in development, the features of the real turret may differ from those of the virtual one.
It features:
- a 80 x 60 pixels resolution
- a horizontal view field of 60 degrees.
- a vertical view field of 60 degrees.
- a 24 bit color depth.
Note: this is a top turret.
Ok

Figure 4.5: The real K5300/K6300 turret

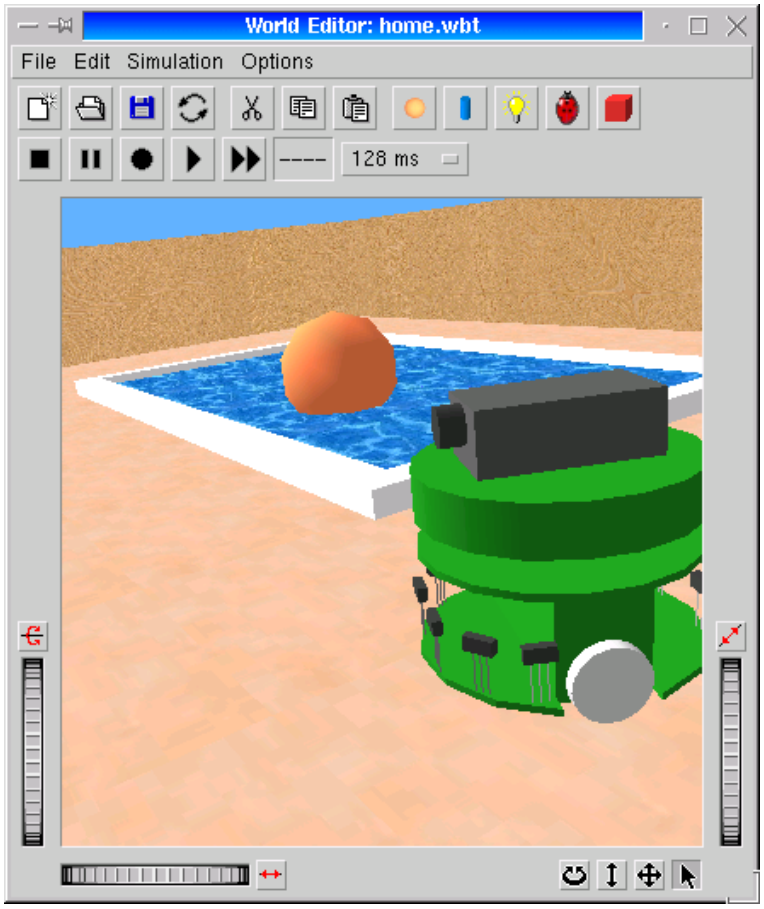

Figure 4.6: The simulated K6300 turret

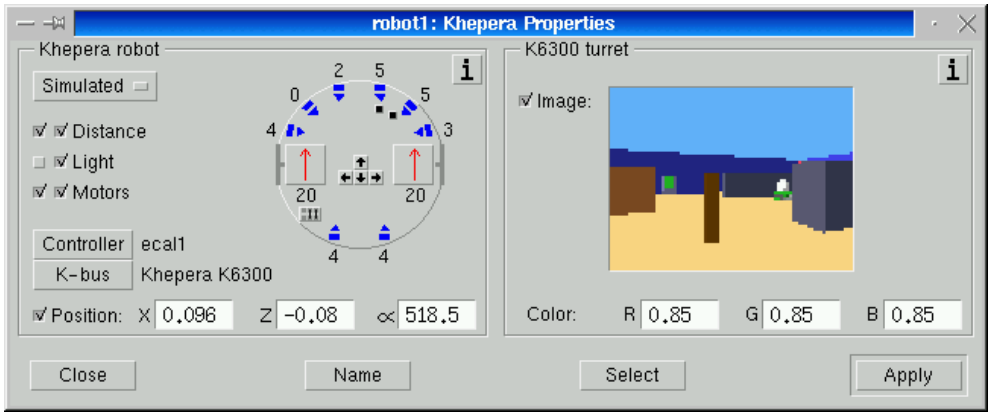

Figure 4.7: The robot window for the K6300 turret

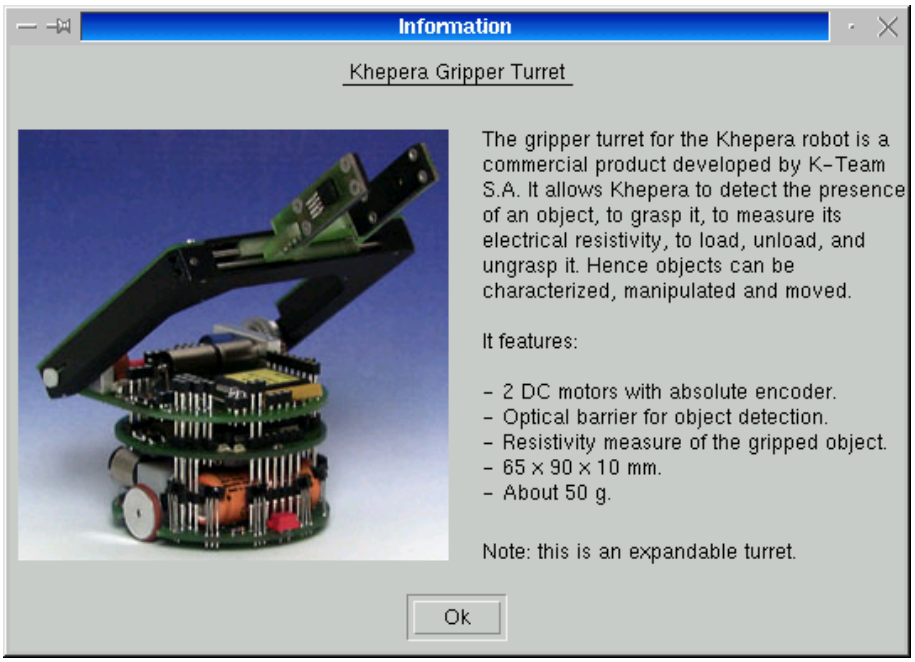
Information
Khepera Gripper Turret
The gripper turret for the Khepera robot is a commercial product developed by K-Team S.A. It allows Khepera to detect the presence of an object, to grasp it, to measure its electrical resistivity, to load, unload, and ungrasp it. Hence objects can be characterized, manipulated and moved.
It features:
- 2 DC motors with absolute encoder.
- Optical barrier for object detection.
- Resistivity measure of the gripped object.
- 65 x 90 x 10 mm.
- About 50 g.
Note: this is an expandable turret.
Ok

Figure 4.8: The real gripper turret

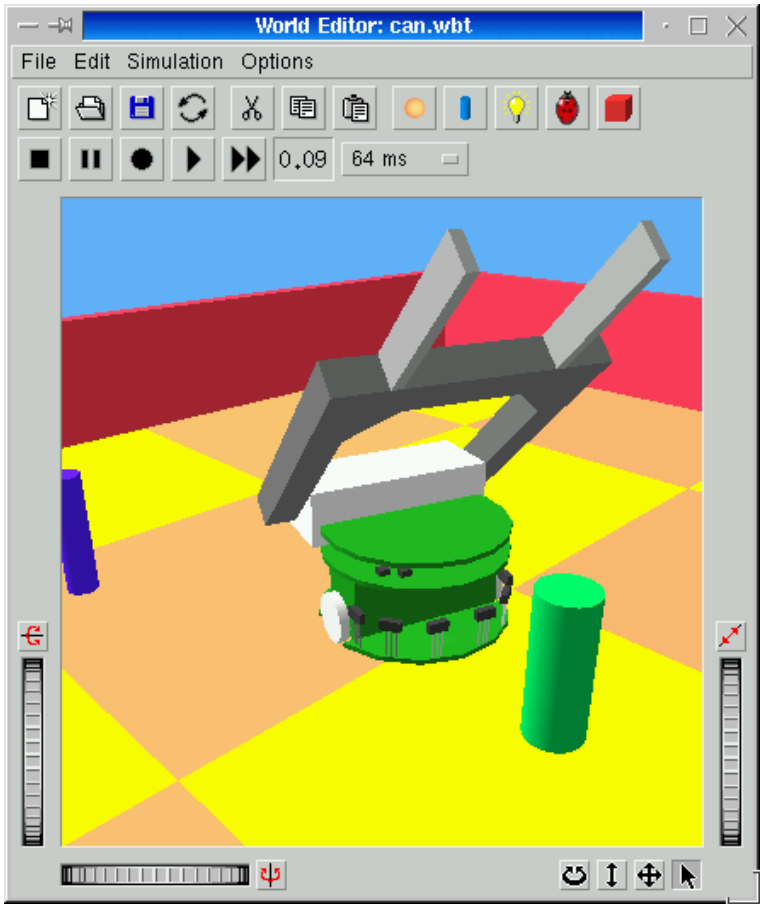

Figure 4.9: The simulated gripper turret

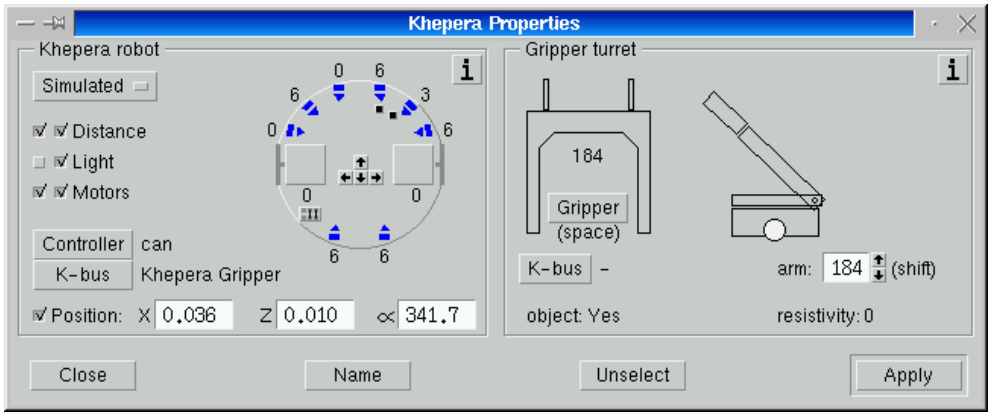

Figure 4.10: The robot window for the gripper turret

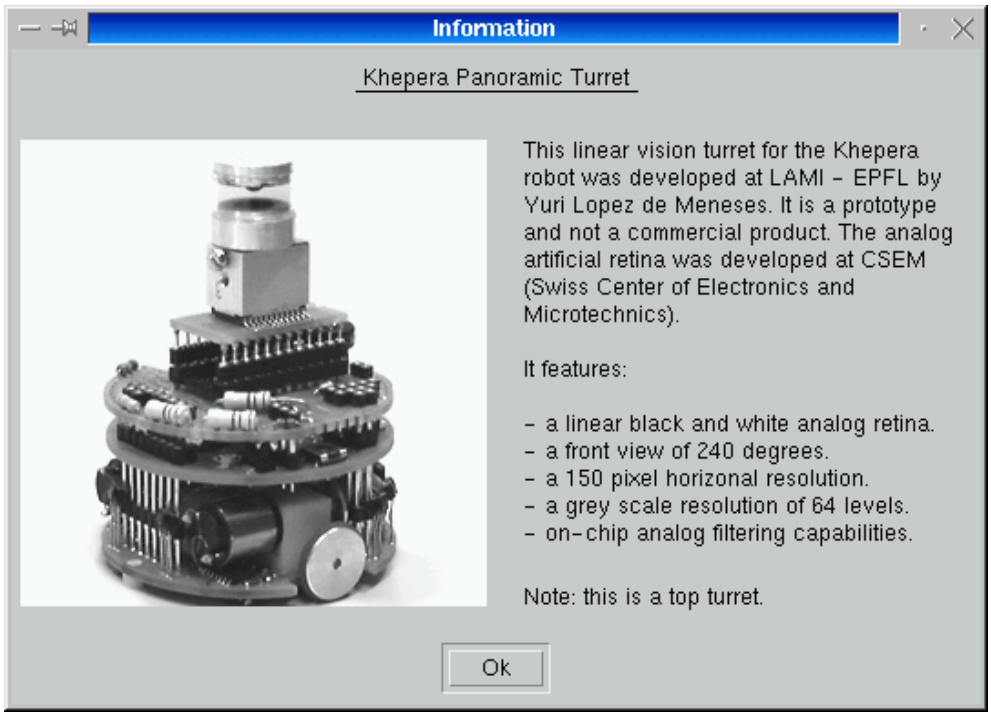
Information
Khepera Panoramic Turret
This linear vision turret for the Khepera robot was developed at LAMI – EPFL by Yuri Lopez de Meneses. It is a prototype and not a commercial product. The analog artificial retina was developed at CSEM (Swiss Center of Electronics and Microtechnics).
It features:
– a linear black and white analog retina.
– a front view of 240 degrees.
– a 150 pixel horizonal resolution.
– a grey scale resolution of 64 levels.
– on–chip analog filtering capabilities.
Note: this is a top turret.
Ok

Figure 4.11: The real panoramic turret (prototype)

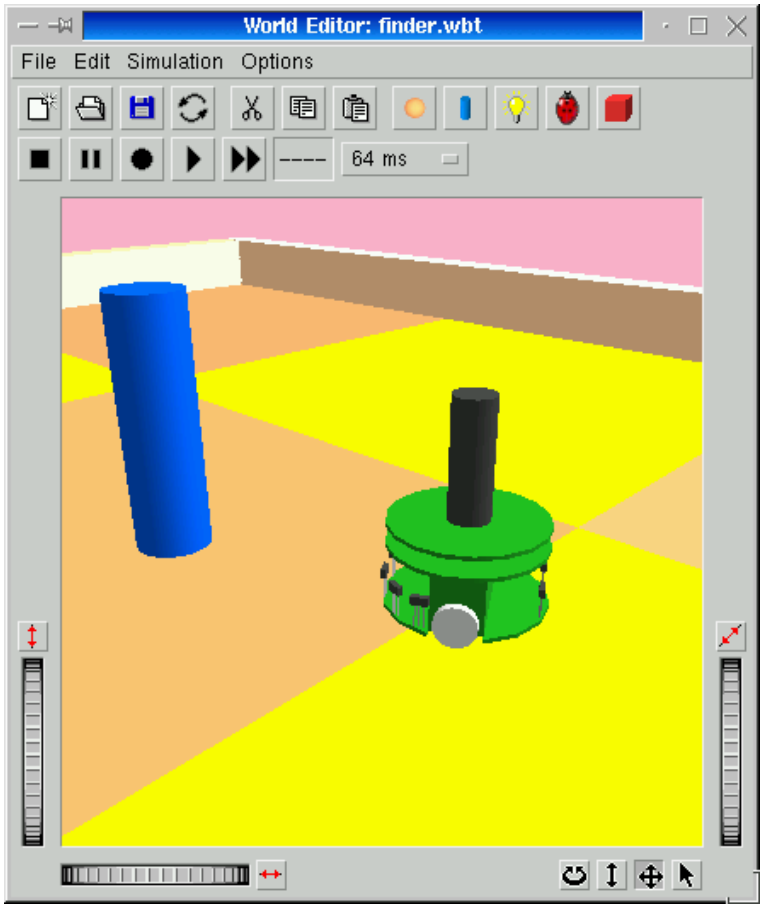

Figure 4.12: The simulated panoramic turret

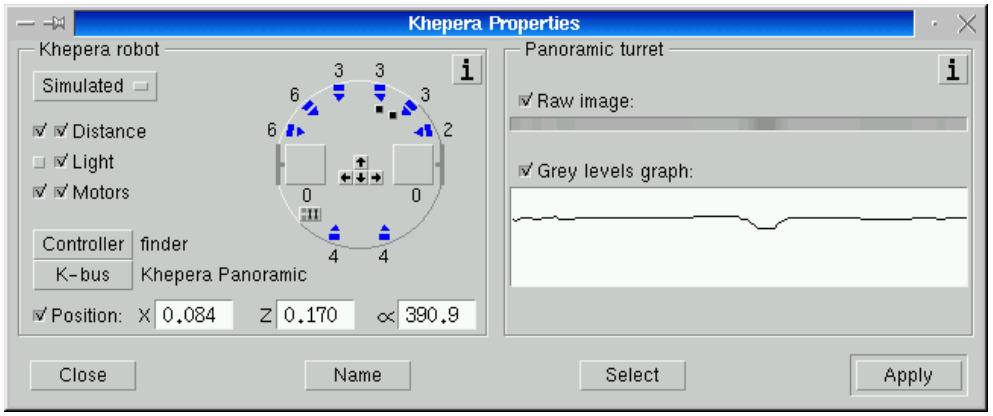

Figure 4.13: The robot window for the panoramic turret

Chapter 5

Programming with the Supervisor API

The Supervisor API is very useful to control experiments, to record interesting data like sensor information, robot trajectory, or any statistics, to introduce abstract sensors, to make the robots communicate with each other, etc.

5.1 EAI: External Authoring Interface

The External Authoring Interface API allows the supervisor controller to read and edit the world loaded in the simulator. This API is quite similar to the VRML'97 External Authoring Interface allowing Java applets to communicate with VRML scenes within a Web browser. In our case, the supervisor controller process is able to communicate with the simulator in a similar way.

5.1.1 Getting a pointer to a node

Each object in the world is called a node (like VRML'97 nodes). Each node can have specific name, called DEF name, which is given by the designer of the world. Such a DEF name can be defined from Webots by clicking on the `Name` button in the properties window of an object. This DEF name can then be used by the supervisor process to get a pointer to that node by issuing the following function call:

```
#include <eai.h>

eai_node pointer_to_my_node;

pointer_to_my_node = eai_get_node(''MyNode'');
```

5.1.2 Reading information from the world

The EAI functions `eai_get_xxx` (where `xxx` can be replaced by `position`, `orientation`, `color`, etc.) allow the supervisor process to read informations on the position, orientation, color, etc. of a number of nodes in the world. The first (and sometimes the only) argument of the `eai_get_xxx` functions is always a pointer specifying the node from which we want to read information. See the reference manual for a complete list and description of the different `eai_get_xxx` functions.

5.1.3 Editing the world

5.1.3.1 eai_set_xxx

The EAI functions `eai_set_xxx` (where `xxx` can be replaced by `position`, `orientation`, `color`, etc.) allow the supervisor to change the position, orientation, color, etc. of a number of nodes in the world. It is for example possible to change dynamically the controller of a robot by calling the `eai_set_controller` function with the name of the new controller as an argument. The first parameter of the eai_set_xxx functions is always a pointer specifying the node which is going to be altered. The following arguments depend on what kind information need to be altered. For example, changing the color of a node will require to pass a pointer to the node as the first argument and the red, green and blue levels as the second, third and fourth argument. See the reference manual for a complete list and description of the different `eai_set_xxx` functions.

5.1.3.2 eai_delete_node

The EAI function `eai_delete_node` is useful to delete nodes from the world. The is currently no way to create new nodes from the supervisor.

5.1.3.3 eai_refresh_world

If the simulator is not running in `Run` mode, i.e., not updating regularly the scene, then the changes performed with a `eai_set_xxx` function will not be visible in the scene until the display is updated. In order to force to refresh the display of the world, the supervisor process can use the `eai_refresh_world` function.

5.1.4 Sending messages to the robots

5.1.4.1 Overview

The only way to control Alice robots in Webots is to send them messages corresponding to behaviors. In order, to proceed, it is necessary to get a pointer to the Alice robot we want

Command	Identifier	Number of bytes
ALICE_FORWARD	1	1
ALICE_BACKWARD	2	1
ALICE_RIGHT	3	1
ALICE_LEFT	4	1
ALICE_SENSORS_OFF	5	1
ALICE_SENSORS_ON	6	1
ALICE_STOP	8	1
ALICE_FOLLOW_RIGHT_AND_TURN	9	1
ALICE_FOLLOW_LEFT_AND_TURN	10	1
ALICE_FOLLOW_RIGHT	11	1
ALICE_FOLLOW_LEFT	12	1
ALICE_STEP_RIGHT + parameter	13	2
ALICE_STEP_LEFT + parameter	14	2

Table 5.1: Alice Messages

to control by using the `eai_get_node` function. Then, the supervisor controller can send it messages using the `eai_write_stream` functions.

The same principle can be used to send messages to the Khepera robot. However, in this case, you will have to program your Khepera controllers so that it handles the messages send from the supervisor.

5.1.4.2 Messages for the Alice robot

The messages sent to the Alice robot must be composed of one or two bytes as described in the table 5.1.

Most of these commands are self explanatory. However, a couple of them need more explanations.

The `ALICE_SENSORS_OFF` and `ALICE_SENSORS_ON` allows the supervisor to turn off and on the obstacle avoidance in the `ALICE_FORWARD` behavior. By default, this behavior performs obstacle avoidance.

The `ALICE_STEP_RIGHT` and `ALICE_STEP_LEFT` commands need an extra byte parameter which indicate the number of steps to be performed. This byte needs to follow immediately the first command byte in the message.

5.1.4.3 Encoding

For compatibility issues with the real robot, each message must be encoded in the following way: The first byte contains the robot identifier and the associated command. Optionally a second byte

contains a parameter. The first byte is organized as described in figure 5.1.

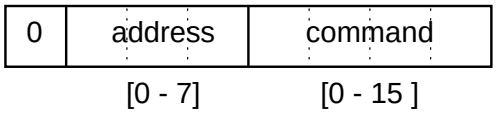

Figure 5.1: Command byte for the Alice robot

The parameter byte when used, contains an integer value coded on the 7 less significant bits (LSB) of the byte as depicted in figure 5.2.

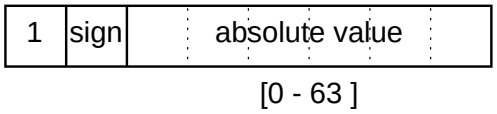

Figure 5.2: Parameter byte for the Alice robot

The sign bit has the following meaning: 0 means positive and 1 means negative. Hence, the parameter value range ranges from -63 to +63.

The example of a supervisor controller for Alice, described in this manual, is available within the Webots package.

Chapter 6

Using GUI: the Graphical User Interface API

Robots controllers as well as the supervisor controller can take advantage of using a simple yet powerful graphical user interface: the GUI, standing for Graphical User Interface. This API (Application Program Interface) allows controllers to open windows, to display texts, graphics and widgets, to handle events coming from the user, like mouse moves, key pressing, etc. It is very convenient to improve the interactivity between the user and the simulation.

This chapter is a tutorial for understanding and using the GUI. However, it does not provide an exhaustive description of this API. See the Reference Manual for an exhaustive description of the GUI.

6.1 Basics

6.1.1 Include file and library

In order to use the GUI, your controller programs will have to include the `gui.h` file which stands in the `webots/include` directory. No additional library is required at link time since the GUI is fully contained into the `Controller` library. So, here is the first line of code to get started with the GUI:

```
#include <gui.h>
```

6.1.2 GUI objects

Different kinds of objects can be used to create a graphical user interface. They are classified into five categories:

- windows: `window`
- gobs (graphical objects): `arc`, `image`, `label`, `line`, `rectangle`
- widgets: `button`, `checkbox`, `popup`, `textfield`
- invisible objects: `pixmap, timer`
- utility objects: `color`, `event`, `screen`

6.1.3 Constants

In the GUI, constants are all uppercase starting with the prefix `GUI_`. Constants are defined for the colors, the events, the keys, etc.

6.1.4 Functions

In the GUI, all function names start with the prefix `gui_`. This prefix is generally immediately followed by the name of an object (like `window`, `rectangle` or `button`), then an underscore: _, then an action associated to this object (like `new`, `delete`, `change_color`, or `get_value`). For example, the function `gui_rectangle_change_color` is used to change the color of a rectangle.

6.1.4.1 Constructor functions

Constructor functions are used to create new instances of objects. They look like the following prototype: `gui_xxx_new` where `xxx` is the name of an object to be created. They apply to all kinds of object except utility objects. A constructor function take a number of arguments used to define the object. See the Reference Manul for a complete description of the various constructor functions.

Note: For a gob or a widget, the first argument is always the handle of the window in which it is created while the second and third arguments are always its x and y coordinates in the window coordinates system. Then, the following arguments define more precisely the object and are described in the reference manual.

6.1.4.2 Destructor functions

Destructor functions look like this: `gui_xxx_delete`. Like constructor functions, they apply to all kinds of object except utility objects. The first and only argument of these functions is a handle to the object to be destroyed. A destructor function frees the memory allocated for the object passed as an argument. Moreover, it makes the object disappear from the screen,

or window where it was. After calling the destructor of an object, no further reference to that object can be done otherwise, it might crash your program. So, it is recommanded to set the corresponding handler to `NULL` immediately after calling a destructor function, unless you are sure that this handler can no longer be used:

```
gui_pixmap p
...
  p = gui_new_pixmap(...);
  ...
  gui_pixmap_delete(p);
  p = NULL;
```

6.1.4.3 Read functions

Read functions allows you to read the properties of an object. They apply to any object. These functions look like this: `gui_xxx_get_yyy` or `gui_xxx_is_yyy` where `xxx` is the name of an object and `yyy` is the name of the property to be read. The `gui_xxx_is_yyy` functions always return a boolean value whereas the `gui_xxx_get_yyy` may return any type. The first argument of these functions is a handle to the object. Other arguments may be used to retrieve some values or specify more precisely the read request. All the read functions are described in details in the Reference Manual.

6.1.4.4 Write functions

Write functions allows you to alter the properties of an object. Except for utilitiy objects, the first argument of a write function is a handle to the object. Write function can have various forms, including, but not limited to, the following:

- `gui_xxx_set_yyy`
- `gui_xxx_show`,
- `gui_xxx_hide`,
- `gui_xxx_enable`,
- `gui_xxx_disable`,
- `gui_xxx_activate`,
- `gui_xxx_desactivate`,
- `gui_xxx_change_yyy`, etc.

where xxx is the name of the object and yyy (if any) is the name of the property to be changed. Usually the gui_xxx_set_yyy functions will change the property yyy of an object xxx without redisplaying it, whereas the gui_xxx_change_yyy function will change the property yyy of an object xxx and redisplay it. For example, the gui_rectangle_change_color will change the color of the specified rectangle and redisplay it.

6.2 Getting started

The very first thing to do, when designing a graphical user interface, is to create a first window. This can be achieved by issuing a gui_window_new function:

```
gui_window my_window;
...
  my_window = gui_window_new(‘‘My First
Window’’,10,10,200,100);
```

Note: the controller program must always be initialized first, that is the xxx_live function (where xxx may be supervisor, or khepera) should be called before attempting to use any of the GUI functions.

Now, in order to add some text in this window, we will use the following function:

```
(void)gui_label_new(my_window,20,20,’’Hello world!’’);
```

This function returns a handle to the new label object it has created, but since we don’t need it, we will just ignore the return value of this function by casting it to void.

After this stage, your window exists and contains some text label, but it is not visible on the screen. In order to make it appear, you will have to use another function:

```
gui_window_show(my_window);
```

Now, we have a very simple yet complete graphical user interface working within a controller program.

6.3 Editing, adding and deleting gobs

6.3.1 Editing gobs

In order to make this simple graphical user interface more useful, it may be interesting to dynamically change the text of the label we just created according to the internal state of the controller program. This can be achieved by changing a little bit the original program, so that we keep a handler to that label object:

```
gui_label my_label;
...
  my_label = gui_new_label(window,20,20,''I am happy
:)'');
```

Then, whenever during the execution of the controller, the internal state may be changing and we might need to change the text of the label:

```
  if (state==SAD) gui_label_change_text(my_label,''I am
sad :(‘‘);
```

Many write functions allow to change the properties of different objects. Their usage and behavior is explained in the Reference Manual.

6.3.2 Adding gobs

Adding gobs is as simple as this:

```
gui_rectangle my_rectangle;

my_rectangle =
gui_rectangle_new(my_window,10,40,30,50,GUI_RED,true);
```

According to the Reference Manual, this will create a red filled rectangle at location (10,40) with a width of 30 pixels and a height of 50 pixels. This rectangle will be immediatly visible. See the Reference Manual for creating other kinds of objects.

6.3.3 Deleting gobs

If ever our label gob is not any more needed, it is possible to delete it, so that it disappears from the window:

```
gui_label_delete(my_label);
my_label=NULL;
```

It is also possible to delete the window itself if is not any more needed:

```
gui_window_delete(my_window);
my_window=NULL;
```

Note: if a window contains some objects (gobs or widgets) which have not been previously deleted, those objects are automatically deleted with the window. Further reference to them may produce a crash of the controller program. Hence we should add the following line for sanity:

```
my_rectangle=NULL;
```

6.4 Working with widgets

Widgets are a bit more complicated than gobs since they provide user feedback to the controller program. It is possible to create widgets the same way as for gobs. However, for handling widgets input, some additional code has to be implemented. Widgets include buttons, checkboxes, popup menus and textfields as shown in figure 6.4.

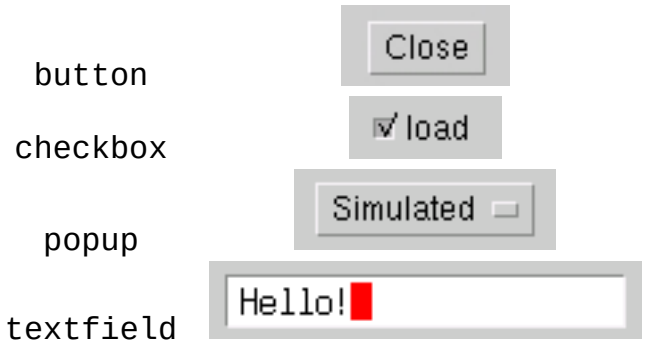

Figure 6.1: Widgets available in the GUI

6.4.1 Events

Events are well known to any programmer dealing with graphical user interfaces. The event model in the GUI is very simple to understand and use. Different kinds of event in the GUI are listed here, all of them, except TIME_UP, are a direct consequence of the user action with the mouse or keyboard:

- `GUI_MOUSE_DOWN`
- `GUI_MOUSE_UP`
- `GUI_MOUSE_MOVE`
- `GUI_KEY_DOWN`
- `GUI_KEY_UP`
- `GUI_WINDOW_CLOSE`
- `GUI_WINDOW_RESIZE`
- `GUI_TIME_UP`
- `GUI_WINDOW_ENTER`
- `GUI_WINDOW_LEAVE`

6.4.2 Callback function

The callback function is a function you will write within your program and declare to the GUI, so that it is called whenever an event occurs:

```
void my_callback()
{
  ...
}

void main()
{
  ... /* GUI settings: create a window with objects in-
side */
  gui_event_callback(my_callback);
  for(;;) supervisor_step(64);
}
```

It is very important that a controller calls a `xxx_step` function in a loop after seting the callback function since this callback function will be called from the `xxx_step` function.

If an event occurs during the execution of the `xxx_step` function, your callback function is called and it has to retrieve some information about the event to update your program, or do whatever you want to do upon reception of events. To achieve this, a number of functions can be used within the callback function:

```
gui_event_get_type
gui_event_get_info
gui_event_get_widget
gui_event_get_window
gui_event_get_mouse_x
gui_event_get_mouse_y
gui_event_get_key
gui_event_get_modifier
gui_event_get_timer
```

In your callback function, you may first want to know what kind (type) of event occured and you will use `gui_event_get_type` to achieve this. To distinguish between widget events and other events, you can use `gui_event_get_info` function. Then, the other functions will provide you with more detail about this event. For example, if the `gui_event_get_info` tells you that a widget event occured, the `gui_event_get_widget` will return a pointer to the widget that caused that event. Note that all functions are not applicable for any type of event,

but rather they are specific to some types of event. A complete description of these functions is available in the Reference Manual.

As an example, to check if a button was pressed by user the callback function has to look like the following:

```
void my_callback()
{
  gui_widget w;

  if (gui_event_get_info()==GUI_WIDGET_EVENT)
  {
    w = gui_event_get_widget();
    if (w==my_button) printf(‘‘my button was
pressed!\n’’);
    else printf(‘‘Another widget was activated!\n’’);
  }
}
```

The callback function can also use widget functions to retrieve information from a widget, like the text contained in a textfield just edited by the user. Moreover, it is useful that the callback function call other functions to update the state of global variable, display some information to the user, etc. However, the callback function should never call the `xxx_step` function directly or indirectly (that is by calling a function [which calls a function, which call a function, etc.] which calls the `xxx_step` function). If this occurs, then, an undefined behavior will be observed and the controller will probably crash.

6.5 Going further

The examples provided within the simulator package can be a good starting point to understand the possibilities of the GUI. Moreover, the Reference Manual covers some issues not discussed here like timers.

Chapter 7

Advanced Webots programming

7.1 Hacking the world files

7.1.1 Overview

The world files end up with the `.wbt` suffix. They lie in the `worlds` directory. These files obey Webots file format which is an extension of a subset of the VRML97 language. VRML97, standing for Virtual Reality Modeling Language, is an official standard, widely used for 3D on the World Wide Web. It features animation and programming capabilities but it is not powerful enough to model virtual robots equipped with realistic sensors. To view such 3D scenes and navigate through them, you currently need to add a VRML97 plugin, as you might to your favorit web browser.

Webots file format was implemented as an extension of a subset of VRML97. Hence, 3D scenes and animations produced by Webots could easily be ported to VRML97 to be published on the World Wide Web.

You can read and edit the world files with a standard text editor. However, if the file is corrupted because the syntax is not respected, Webots may crash when trying to load the file. Please note that not all VRML97 nodes are implemented. Moreover, within the implemented nodes, not all fields are implemented. So, don't use nodes or fields that are not implemented, this would cause Webots to crash.

The Reference Manual gives a complete list of nodes and associated fields currently supported in the Webots file format.

7.1.2 Robots and Supervisors

Webots file format allows storage of robots and supervisors as parts of environments. Robots are defined by their position, configuration, and controller code. Supervisors are defined only by their controller code. A world file should contain no more than a single supervisor.

7.1.3 Textures

A subset of the VRML97 texture nodes is supported in Webots 2.0. You might make use of them to set any image as a texture for a specific shape. Texture files must be in PNG format, and their location must be specified relative to the world directory (usually, all textures are stored in the `textures` subdirectory of the `world` directory).

A textures image should have a minimal size of 64 x 64 pixels. Moreover, its width and height should be a power of two, otherwise, it will be truncated to the first inferior power of two. For example, if a texture image is sized 150 x 300 pixel, it will be truncated to a 128 x 256 image before proceeding the rendering.

Textures can be applied to `Wall`, `Can`, `Cylinder`, `Box` and convex `IndexedFaceSet` nodes. Please refer to the reference manual to see in detail which are the supported nodes and corresponding fields.

7.2 Using external C/C++ libraries

All the sample programs in the Webots distribution are C programs which don't rely on external libraries. However, it is possible to develop C++ programs as well and to make use of external libraries. In order to do so, you will have to modify your `Makefile` files. Webots include files are designed to support C++ as well.

7.3 Interfacing with third party software

C and C++ are not the only programming languages in the world of computers. Hence, you may want to use another programming language to drive your robots or supervisors. Lisp, Java, Matlab or whatever language can be used within Webots with only a small development effort. Moreover, interfaces to scientific data display software like gnuplot are also possible and can be achieved the same way.

Webots needs to dialog with a C or C++ based controller program which is linked with the controller library. However, this controller program can be just an interface to the third party software you want to use.

7.3.1 Using a pipe based interface

The communication between this interface controller and the third party software can be achieved through the use of pipe files as long as the third party software supports reading from and writting data to a pipe file (which is however very common). Usually, you will want to create two pipe files, one for the input data coming from the interface controller and going to your third party

program, and another for the output data, coming from your third party software and going to the interface controller.

The development of such an interface can be divided into two stages:

1. Develop the interface controller in C. This is the easy part since the example `interface.khepera` is prodived within your webots package
2. Develop the third party software pipe interface.

In order to proceed on the third party software side, you have to look in your user manual

7.3.2 Using other Inter Process Communication systems

Although they are not covered in the manual, any other Inter Process Communication (IPC) system could be used to achieve the same purpose. The most interesting, though, would be a socket-based network interface, so that you can distribute controller computation over a network of computers. This could prove to be very useful for multi-agent simulations using computer expansive controllers.

Chapter 8

Troubleshooting

This chapter covers a number of known issues that may arise when using Webots. Please read it carefully. It can help for most common problems with Webots. However, if the problem remains, please, send a bug report to Cyberbotics.

8.1 Common problems and solutions

Problem: Webots sometimes leaves some controllers or supervisors alive even after quitting. Such controllers or supervisors are independent processes using system resources. It might be useful to destroy them in order to release the resources they use.

Solution: The useless controllers and supervisors should appear in the list of processes by issuing a `ps` or a `top` command. Then, you will get their `pid` (process identifier) and will be able to destroy them with a `kill` command.

Problem: The real robot doesn't work via the serial link. Webots says "Unable to open serial port".

Solution: Check that the permissions of the serial device files are set appropriately. These files must be readable and writable by all the users. For example, to check `/dev/cua0`, you can do:

```
ls -l /dev/cua0
crw-rw-rw-  1 root   uucp    5,  64 Aug 27 16:59 /dev/cua0
```

These are the right permissions. If they are different, you can change them by logging on as root and typing:

```
chmod a+rw /dev/cua0
```

8.2 How to I send a bug report ?

If you find a bug in the Webots software, or have a problem which is not covered within the documentation and the examples, than you should send a bug report to Cyberbotics. In order to do so, write an e-mail to `support@cyberbotics.com`. This e-mail must contain the following information:

1. Your name.
2. The version of Webots you use.
3. A complete description of your system configuration: machine, operating system, and eventually additional hardware like 3D graphics board.
4. A complete description of the bug allowing us to reproduce it step by step.
5. Optionally, some material allowing us to reproduce the bug (i.e., the source code of a controller program, a world file, etc.).